Action Systems, Determinism and the Development of Secure Systems

Thesis

UNRESTRICTED

Action Systems, Determinism and the Development of Secure Systems

PhD Thesis

Jane Sinclair

Computing Department
Faculty of Mathematics and Computing
The Open University

10th February 1998

Date of submission: 15th October 1997
Date of award: 9th March 1998

Action Systems, Determinism and the Development of Secure Systems

Jane Sinclair

Computing Department, The Open University.

PhD Thesis. February 1998

Abstract

This thesis addresses issues arising in the specification and development of secure systems, focusing in particular on aspects of confidentiality. Various confidentiality properties based on limiting the allowed flows of information in a system have previously been proposed. These definitions are reviewed here and some of the problems inherent in their use are outlined. Recent work by Roscoe [106] has provided information flow definitions based on restricting the allowed nondeterminism within the system. These properties are described in detail, with a range of examples provided to illustrate their use.

This thesis is concerned with providing a new, pragmatic approach to the development of secure systems. Action systems are chosen as a notation which incorporates both direct representation of system state useful for effective system modelling and the succession of events in a system essential for representation of information flow properties. A definition of nondeterminism and formulations of the deterministic security properties are developed for action systems. These are shown to correspond to the original CSP event-based definitions.

The emphasis of this work is on the practical application of theoretical results. This is reflected in the case studies in which the preceding work is applied to realistic development situations. This allows the strengths and weaknesses of both the deterministic security conditions and the use of action systems to be assessed. The first study investigates security constraints applied to a distributed message-passing system. Ways of specifying security conditions and the effects of including them at different levels are explored. The second case study follows through the specification and refinement of a distributed security kernel. A technique for the simplification of security proofs is introduced.

Acknowledgments

Thanks to my supervisor Darrel Ince for his support and encouragement. Thanks, too, to the Open University which provided the opportunity for my studies as it has for many other mature students.

A number of friends have read and provided comments on parts of this thesis. For their time and effort and for their comments which have been a great help to me, my thanks to Michael Butler, Jon Hall, Jeremy Jacob and Ian Livingstone.

Finally, special thanks to my husband Ian who understands the trauma of doing a PhD, and to my children Helen and Ben who help to keep things in perspective.

This work was supported by the EPSRC.

Contents

List of Figures

Chapter 1

Introduction

The need to provide secure computer systems has now been a topic of concern for nearly thirty years. Once the use of computers progressed from single users directly accessing stand-alone machines, the need for security controls which would apply specifically to computer systems became apparent. Since then, there has been much research into computer security. Developments in the use of computers, such as the Internet and indirect accessing, have presented further challenges. As a clearer idea of security needs has emerged, many security-related products have become widely available (for example, for encryption and for secure communication) and standards have been set down for the evaluation and accreditation of such products. The demand for improved security and greater accountability continues to grow, as do the challenges faced by the providers of secure systems.

The work of this thesis considers one particular branch of research: that of providing a formal definition of security properties for the specification and development of secure systems. It attempts to incorporate such a definition in a pragmatic approach to the practical development of secure systems. The aim of this thesis is to provide a context for some of the theoretically-appealing formal security definitions which facilitates their application to the development of general-purpose secure systems. Having formulated security properties in such a way, it is then possible to investigate the strengths and weaknesses of the approach and of the formal properties themselves. To achieve these aims, this work first reviews and compares existing definitions.

Properties based on determinism are seen to have a number of theoretical advantages and interpretations of these properties are given in a general-purpose specification notation. To assess the suitability of the definitions and of the development approach two case studies are presented. Both represent aspects of real computer systems and are on a large enough scale to test out the security properties and the development approach. This chapter sets the context for that work and outlines the structure of the thesis.

1.1 Confidentiality

Threats to computer security have traditionally been divided into three categories, concerning confidentiality, integrity and denial of service. This thesis addresses the first of these. Confidentiality (otherwise referred to as privacy, secrecy or non-disclosure) encapsulates the general idea that information may be disclosed only to those who are entitled to see it. In order to judge the security of a system there must exist a definition of security against which the system can be assessed. Much research in computer security has been directed towards providing a suitable formal definition of confidentiality.

Although the idea of confidentiality is a simple and intuitive one, providing a theoretical basis for the confidentiality of computer systems has proved notoriously difficult. Proposed definitions which at first appeared promising have later been found to be deficient in various respects or to be applicable only in certain, restricted circumstances. Revised definitions moved away from the earlier, straightforward approaches resulting in less intuitive and sometimes complex conditions (see Chapter 2). As a result, there currently exist many alternative proposed properties for confidentiality. The diversity of the definitions and the variety of notations in which they are proposed complicates the task of comparison and assessment.

1.2 Information flow

Since the work of Goguen and Meseguer [43, 44] in the early 1980s, much research into providing confidentiality properties has focused on information flow. Using this approach, a security policy is constructed by specifying the allowed flows of information within a system. In order to decide whether such a requirement is satisfied or not there must be some criterion setting out what constitutes information flow. Goguen and Meseguer described a property, known as noninterference, which can provide the underlying theory for an information flow policy. For user $u1$ to be noninterfering with user $u2$, no actions of $u1$ may be allowed to affect the view of $u2$.

Since the early work on noninterference, many different information flow definitions have been proposed. These have generally sought to extend the original ideas to a wider context (for example, to nondeterministic systems) or to include other desirable properties (such as the requirement that the composition of two secure systems should itself be secure). Some have been regarded as a generalised form of the original idea of noninterference, with others being sufficiently different to merit new labels. Information flow definitions have been presented in a number of different notations, some of the earlier work being couched in terms of state-transition machines with some more recent authors favouring an event calculus such as CSP.

The number of different conditions and notations can obstruct the comparison and assessment of these properties. Many were developed to address a specific concern and the exact range of situations each covers is not immediately apparent.

Recent work by Roscoe and Woodcock [107, 110] has provided further insight by noting the relationship between information flow and nondeterminism. The research of this thesis builds on that approach by extending the deterministic properties to systems where the representation of system state is regarded as being of equal importance to the succession of events.

1.3 Security policies and the development process

Perhaps the difficulty of interpreting and applying the information flow properties is one reason why, despite the large amount of research in this area, little use has so far been made of them in industry. Such strong properties are certainly not required in all cases, but there is evidence (see Chapter 3) that even formal security developments with strict confidentiality requirements have opted for better understood but less effective access control approaches, and for the use of a general-purpose state-based notation such as Z. In industry more use appears generally to be made of state-based notations than of event-based ones, yet these do not so naturally accommodate the event analysis which has proved so useful for describing information flow.

Real security policies are rarely a straightforward application of a single definition. At best, they might be a combination of different approaches; at worst a complex set of requirements showing little cohesion. Formal approaches to security need to be flexible enough to accommodate such a range of applications. It is also the case that a clear indication of how such approaches fit into an overall development method would aid their application.

1.4 Overview

This thesis examines existing information flow properties and shows how deterministic security conditions can be defined for state-based specification using action systems. It also considers wider issues concerning the development of secure systems with action systems. Chapter 2 presents a survey of confidentiality properties, describing the information flow approach based on noninterference and its variations. The chapter concludes with an overall assessment of these conditions which leads on to a description of the CSP deterministic security properties in Chapter 3. The discussion is motivated by considering the relationship between security and determinism. The properties are illustrated by a number of CSP examples which show how they are

used and highlight the differences between them and other information flow properties. The final section of this chapter considers desirable features of any development method for secure systems. This is discussed in terms of the deterministic security properties and their use.

Chapter 4 introduces action systems as providing a suitable notation for the development of secure systems. In an action system, state and events are given equal consideration. The state can be described using existing notations, such as Z or specification statements. Action systems can be viewed in CSP terms by giving them a failures-divergences semantics, and this correspondence is described here. The central concept for the deterministic security conditions is determinism itself. This is defined for action systems and its soundness with respect to the CSP definitions is proved via the failures-divergences correspondence.

Chapter 5 makes use of the deterministic definitions of Chapter 4 in defining deterministic security properties for action systems. Again, soundness is proved with respect to the CSP definitions. Examples are used to show how these properties work for action systems and to show the relationship with CSP. This chapter also begins to consider how different security policies may be constructed using the basic building blocks of determinism and abstraction. The chapter concludes with a worked example of a multiple stack system which is proved secure with respect to one of the deterministic security conditions.

Chapter 6 considers the refinement of action systems, showing how simulation can be used to refine both state and events. Examples are given, and two different versions of a key server specification are used to illustrate both forwards and backwards simulation. Action systems are particularly suited to the state-based description of distributed systems since a single action system can be refined to a parallel decomposition of action systems. Each component system can itself be further refined and decomposed. Further, when a deterministic security property is used this can be verified at the top level and is then guaranteed to remain true at each subsequent level of refinement. A third approach to the key server specification shows how parallel

refinement works.

Chapter 7 presents a case study in the area of network security. An action system specification of network communication is used. Different approaches to adding security constraints are tried and their implications considered. The study addresses general issues of secure specification, such as how encryption should be represented and how the behaviour of an intruder can be viewed.

A second, more specific case study is given in Chapter 8. This takes as its starting point a description of an existing distributed security kernel. By describing the system as an action system, the necessary structure for considering information flow properties is emphasized. Applying the deterministic security properties to this example illuminates both the strengths and weaknesses of these conditions. A security policy for the system is formed by combining a deterministic property with additional security requirements placed on the state of the system. The chapter also investigates approaches to proof which make the task more manageable.

Finally, Chapter 9 reviews the work and considers related research in this area. Conclusions are drawn and future research directions arising from the current work are suggested.

Action systems notation is described in the text as it is needed. CSP, Z and weakest precondition notation are summarised in the Appendices.

Chapter 2

Confidentiality, information flow and noninterference

This chapter provides an introduction to confidentiality properties for computer systems. It describes how confidentiality may be defined by placing limitations on the allowed flows of information within a system and outlines some existing information flow properties. The material included here motivates and informs the discussion of the final section and paves the way for the approach to confidentiality discussed in Chapter 3. A background of more general computer security issues may be found in a number of textbooks, for example Pfleeger [102] and Amoroso [4]. Some interesting details of early computer security history (as well as more recent developments) are given by Russell and Gangemi [115].

2.1 Introduction to confidentiality

One of the earliest approaches to preventing the unauthorised disclosure of information stored on a computer was the use of access controls. Such controls, often combining both mandatory and discretionary components, prevent users from interacting with objects of the system in prohibited ways. Permitted access can be conveniently described using an *object* $\times$ *subject* matrix. This approach was documented by Lampson [74] and extended in the work of Denning and Graham [30, 46] and provides a straightforward and in-

tuitively appealing way of preventing unauthorised disclosure. Descriptions of a number of models based on access matrices are given in Landwehr's survey [77]. A fairly direct translation can be made from the matrix to an implementation, as shown by Jain and Landwehr [67]. Many examples of the access control approach exist and it continues to be a topic both of practical use and theoretical interest (see work by Sandhu [117, 118] for example).

While access control plays a vital part in the implementation of secure systems, the use of access constraints for describing systems at a more abstract level has certain disadvantages. As pointed out by Landwehr [77], with an access matrix approach it is difficult to prove general properties. Harrison et al. [57] showed that for an arbitrary matrix the question of whether an arbitrary access can at some stage be acquired is undecidable. More recent work by O'Shea [100] investigated other problems of complexity in access control systems. Another problem with access control is that a system can obey the specified access constraints and yet information may still be passed illicitly via covert channels. These were recognised by Lampson [75]. A covert channel is any communication channel within a system which can be used (unexpectedly) to transmit secure information in ways which contravene the system's security policy. Unlike overt channels of communication, which could be taken into account by an access control policy, covert channels are not intended paths for transmission of information within a system, and fall outside the scope of access controls. Work carried out on the identification of covert channels and the calculation of their capacity includes that of Millen [89, 90], Kemmerer [71] and Fine [33].

A model of confidentiality based on a particular view of access control was developed by Bell and LaPadula in the early 1970s [11, 12]. This well-known and popular approach introduced a framework for considering the security of a system separate from its functional requirements. The system, often referred to as BLP, is characterised by two central requirements: "no read up" (ensuring that no one can observe an object at a security level which is higher than their own clearance) and "no write down" (no one can alter an object at a security level lower than their clearance). The development of BLP was

a significant achievement, providing a practical approach which is still used in project development today. However, as a general-purpose method for capturing security requirements it has a number of weaknesses. Modification of the basic framework to accommodate differing security policies is not an easy task since a particular view of security is built into the model. Systems conforming to BLP can still harbour covert channels as shown by Millen [89]. Work by McLean [83, 86] demonstrated that systems exist for which BLP's Basic Security Theorem may be proved but which are intuitively insecure. Understanding of the strengths and weaknesses of BLP has provided useful input for later research.

2.2 Information flow

A different approach to capturing the security requirements of a system is to concentrate on the movement of information within the system and to state explicitly what flows of information are allowed (or which are banned). This approach is typically further removed from the concrete realisation of the system than specification of access controls would be. It has the advantage of allowing a high-level, concise statement of the security requirements, and can support mathematical analysis and proof of critical properties. Early work on information flow was carried out by Denning [28, 29] who specified security policies in a lattice of security classes by giving a suitable "can-flow" relation which defined permitted information flows within the lattice. Restriction of information flow is the basis of the frameworks described in the remainder of this chapter.

Information flow analysis provides a way to concentrate on what is required, separating this from the business of how it will be achieved, particularly when used in conjunction with a formal specification of the functional behaviour of the system. This is extremely useful for description and analysis of secure information flow. However, it does not provide an instant solution to the problems of covert channels. Extensive effort may be required to identify such channels and, even then, it may not be feasible to take preventative

measures. On the other hand, information flow analysis does provide a tool which has the potential to analyse unexpected possible flows at different levels and gives developers the opportunity to deal with them.

An information flow policy which details the allowed flows of information in a system must be accompanied by a definition of what constitutes a flow of information. The properties described below all attempt to provide such a basis. A brief account of a number of existing information flow properties is given to provide a context for the development of this approach to confidentiality.

2.3 Noninterference

Noninterference is the best-known information flow property. However, the term does not refer to a single, specific definition, but has been used to cover a number of varying properties described in a variety of notations. These are reviewed in this section. In the following descriptions it will be assumed that, unless otherwise stated, there are two system users, H and L, whose interactions with the system partition those available. This improves the clarity of the descriptions and in most cases can be generalised to multiple users.

High-level user H is said to be noninterfering with low-level user L if no actions of H can affect the view of L. If H is noninterfering with L then no information can flow from H to L. Noninterference properties can be used to express not only standard, multi-level security but many other requirements too.

2.3.1 Basic noninterference

Goguen and Meseguer [43, 44] gave a definition of noninterference for deterministic state machines. In this approach the transition between states is total and functional for each user command. The only information a user can derive from the system is from the output seen by that user after each transition. Let $out(u, t)$ denote the output to u after the sequence of com-

mands t has occurred (starting from the system's initial state). Also let $t \upharpoonright u$ represent t restricted to the commands of u. Then user H is noninterfering with user L if for each sequence t of commands:

$$out(L, t) = out(L, t \upharpoonright L) \qquad [GM]$$

No commands of H can make any difference to what is observed by L, so no information can flow from H to L.

Example 1 Users H and L share a resource which each of them may request. If the request is currently unused it will be granted to the requesting user and further requests denied until the resource has been released by the current owner. Although there is no direct writing of data from one user to the other, information can still flow, since H can interfere with L. (Information can also flow from L to H, but we are not concerned about that). This can be seen, for example, with:

$$t = \langle (request, H), (request, L) \rangle$$

since:

$$out(L, t) = denied$$

but

$$out(L, t \upharpoonright L) = out(L, \langle (request, L) \rangle) = granted$$

In CSP terms, observation of a system is by the set of events offered at each stage. Hoare's Communicating Sequential Processes (CSP) [59, 109] is a widely used notation for the specification of concurrent systems. Events are the basic elements of CSP, with events themselves being given no underlying structure. Each event is viewed as occuring instantaneously. A process is defined by setting out the events in which it may engage and in what order they may occur. Processes interact by simultaneous participation in events common to both. A summary of the CSP notation used is found in Appendix

A. For process P, a user can be conveniently represented by the events in which that user can engage. We are assuming that H and L partition αP. Outputs are themselves events, with an output channel formed by bringing together the separate events for each possible output value. So for an output, only the event for a current possible output value will be offered. This has led to adaptations of the original noninterference property for CSP, such as that given by Ryan [116] which states that, for any trace t, the events offered for L after t must be the same as those offered after $t \upharpoonright L$.

$$(P/t)^0 \cap L = ((P/(t \upharpoonright L))^0 \cap L \qquad [Ryan1]$$

where P^0 is the set of events initially offered by P. A similar definition was given by Allen [3]. It is referred to as extended noninterference and addresses the issue of $t \upharpoonright L$ producing a sequence which is not a trace which would cause $P/(t \upharpoonright L)$ to be undefined.

In contrast to the *GM* property which is defined only for deterministic systems, *Ryan1* and extended noninterference are stated in a notation which allows the presence of nondeterminism. A nondeterministic system is one which can make internal choices which may affect the view of its users. This raises additional questions in the attempt to define noninterference. Although the CSP properties mentioned so far may be applied to nondeterministic systems, the observable effects of internal choice are not taken into account by such properties. Consider the CSP process PN with $H = \{h\}$ and $L = \{l\}$ defined:

$$PN = l \rightarrow (l \rightarrow PN \sqcap h \rightarrow PN)$$

Here, the choice between h and l is an internal one. A property such as *Ryan1* will be satisfied since l may always be offered. However, it is equally possible that l will be refused. This has no effect on the possible traces of the system, but it means that L may observe a refusal, with l only being offered again once h has occurred. Thus L has direct knowledge of H's activity, providing a possible channel for information flow. If this type of interference in a nondeterministic system is of concern to us, then, in CSP terms, the

system needs to be modelled at a level which can describe failures as well as traces. The definition of noninterference will also need to be broadened to take the possibility of internal choice into account .

2.3.2 Noninterference for nondeterministic systems

In addition to *Ryan1* a number of other extended definitions have been given to allow noninterference to be considered for nondeterministic systems. However, as with the two definitions already given, many of these allow the property to be stated in the presence of nondeterminism but do not take into account the special issues concerning nondeterminism as illustrated by process PN above. For example, McCullough's generalised noninterference property [81] basically requires that:

$$(P/t) \setminus H =_T (P/(t^\frown\langle h\rangle)) \setminus H \qquad [McC1]$$

Thus, whether the h action occurs or not, the continued behaviour of the system is the same from L's viewpoint. The hiding operator makes all H events internal to the system. The equality $=_T$ shows that it is equivalence of traces which is under consideration. This property can be used in the context of nondeterministic systems but would not reveal the information flow from H to L in PN above.

Traces are not sufficient to detect the distinction between internal and external choice. In CSP terms, the failures-divergence semantics is needed. There are definitions which incorporate this; such as Ryan's second property [116] which, using CSP refusals, states that for each trace t:

$$\mathit{refusals}(P/t) =_L \mathit{refusals}(P/(t \upharpoonright L)) \qquad [Ryan2]$$

where $=_L$ considers only elements of L in each refusal set.

Another CSP approach to noninterference was provided by Graham-Cumming [47, 48, 49]. Although stated first for traces and essentially equivalent to *Ryan1*, the property is generalised by providing an algebraic formulation which can be interpreted in various semantic models. This allows

failures/divergences to be taken into account if required. This property states that for each trace t:

$$P/t \equiv_L P/(t \upharpoonright L) \qquad [GC]$$

where $\equiv_L$ is the equivalence on processes with all H events suppressed and $P \equiv_L Q$ is defined as:

$$P \,||\, STOP_H = Q \,||\, STOP_H$$

Equality can then be interpreted in the required semantic model. Note that if $t \upharpoonright L$ is not a trace then $P/(t \upharpoonright L)$ is undefined and GC does not hold. For the specific case of process PN as defined above:

$$PN = l \rightarrow (l \rightarrow PN \sqcap h \rightarrow PN)$$

and for the trace $t = \langle l, h \rangle$ then:

$$PN/t = PN$$
$$PN/(t \upharpoonright L) = l \rightarrow PN \sqcap h \rightarrow PN$$

Considering these in parallel with $STOP_H$ gives, firstly:

$$PN \,||\, STOP_H$$

which is equivalent to the process $R1$:

$$R1 = l \rightarrow (l \rightarrow R1 \sqcap STOP_H)$$

and secondly:

$$(l \rightarrow (PN \,||\, STOP_H)) \sqcap STOP_H$$

Any trace of ls is possible for both. Considering failures, $(\langle\rangle, \{l\})$ is a failure of the second but the first will always be prepared to offer l initially. Hence H is not considered noninterfering with L in PN by this definition. GC is similar to the definitions of Ryan when interpreted in the appropriate semantic model.

2.3.3 Inference and noninterference

Jacob [61, 62, 66] carried out much work towards providing a mathematical framework for security properties. Inference functions are used to define what a user with a specified interface might be able to deduce about the rest of the system. The traces of a system P can be projected onto user U by:

$$P \circ U = \{t \upharpoonright U \mid t \in traces\ P\}$$

For user L and observation $tl \in P \circ L$ the infer function is given by:

$$infer\ P\ L\ tl = \{t \in traces\ P \mid t \upharpoonright L = tl\}$$

If L observes trace tl then L knows that the overall system behaviour must be one of the possibilities included in the set $infer\ P\ L\ tl$. This takes into account the possibility that L knows the structure of P. The inference function is used both as a basis for security specifications and as a way of comparing the level of security of two separate systems. A noninterference condition based on Jacob's inference functions was reported by Graham-Cumming [47] as:

$$\forall tl \in P \circ L \bullet \langle\rangle \in (infer\ P\ L\ tl) \circ H \qquad [Jacob]$$

This requires that any trace as seen by L could be possible with no events of H occurring. For the case where H and L partition P this may be stated: for any trace t:

$$t \upharpoonright L \in traces\ P$$

This property is weaker than definition GC, since if $t \upharpoonright L \notin traces\ P$ then $P/(t \upharpoonright L)$ is undefined and so GC cannot hold. That it is strictly weaker is illustrated by the process:

$$P = h \rightarrow STOP \ \Box\ l \rightarrow STOP$$

which satisfies *Jacob* but not CC.

Although inference functions as defined by Jacob allow inferences to be made based on trace behaviours, it would be possible to extend this to an

idea of inferences which include the failures of a system. This would allow observation of refusals to be taken into account.

An approach very similar to the inference function definition of Jacob is the noninference property described by O'Halloran [98]:

$$\forall t : traces\ P \bullet (t \setminus H \in traces\ P) \vee (t \upharpoonright L = \langle\rangle) \qquad [O'Hal]$$

For the case of H, L partitioning P this is equivalent to *Jacob*. For the more general case where events other than those of H and L are possible *O'Hal* is stronger than *Jacob*. Each disjunct of *O'Hal* implies *Jacob*. To see that it is strictly stronger, consider the process PI whose alphabet is partitioned by $H = \{h\}, M = \{m\}, L = \{l\}$, defined:

$$PI = l \rightarrow m \rightarrow PI \ \Box\ h \rightarrow m \rightarrow PI$$

The trace $\langle l, m, h, m\rangle$ does not obey *O'Hal* since removing H events gives $\langle l, m, m\rangle$ which is not a trace. However, every element of $PI \circ L$ is a sequence of ls. This could be inferred by L from a sequence of ls alternating with ms. Thus the *Jacob* property holds for PI.

2.3.4 Composability and perturbations

Another direction in which noninterference definitions have been extended is to try to provide a property which is preserved by system composition. McCullough [80, 81, 82] noted this as a desirable feature and showed that for certain ways of "hooking together" component systems, a confidentiality property such as *McC1* might not be preserved. This motivated further definitions, seeking to restrict the possible influence of high-level inputs by placing stricter constraints on the traces of a system. These definitions feature a distinction between the inputs, outputs and other events in the system. This is different to the various CSP definitions given above in which no separate treatment is given to inputs and output. The "hook-up" security property is described by McCullough [80]:

> We will say that a system is *hook-up secure* if for all traces τ_1, and for all sequences τ_2 formed from τ_1 by adding or deleting high-

level inputs, there is a trace τ_3 such that τ_3 is the same as τ_2 in the constant portion, and differs from τ_2 in the changed portion only in high-level outputs, and such that the first changed output of τ_3 occurs no sooner than the first output in the changed portion of τ_2. [*McC2*]

This is another property given in terms of traces. It is one of a number of such properties where security is dependent upon the occurrence (or absence) of inputs and outputs in particular positions, and is preserved when components are "hooked up" by allowing outputs of one to become inputs to the other. The subtle differences between such definitions can be difficult to characterise, and a property stated in this way would be difficult to verify for a given system. The hook-up property appeared in McCullough's later papers [81, 82] under the name of restrictiveness, with an alternative definition given in terms of state machines. No justification is given to show that the different representations are equivalent, but the idea of the property is that no high-level input can affect the low-level view of a trace. The delay in outputs for τ_3 is the requirement which prevents high-level outputs feeding back to introduce insecure composition. Restrictiveness was given a CSP interpretation by Graham-Cumming [47] and was shown to be strictly weaker than *GC* even for trace equivalence.

The ideas motivating restrictiveness were taken forward by Johnson and Thayer [68] with their definition of forward correctability. This also supports McCullough's "hook-up" result but, taking the view that restrictiveness is stronger than it need be, defines a somewhat weaker property. The definition rests on the deletion and insertion of certain events in a trace. A sequence obtained by deletion or insertion of high-level inputs is called a perturbation. The result of deleting or inserting high-level, non-input events is termed a correction. The definition of forward correctability is given [68] as:

An event system is forwardly correctable iff for any trace α and any perturbation α' obtained by inserting or deleting a single high-level input closely preceding a high-input-free segment γ, there is a correction of α' supported in γ. [*JT*]

Here, an event closely precedes a segment if the two are separated by, at most, a low-level input. A number of different definitions can be obtained in the same way by considering various different perturbations. Some of these were given by McLean and Meadows [84]. Such definitions can be difficult to assess and, as mentioned above, to verify. Unwinding theorems (see Section 2.5) go some way towards alleviating the difficulty of proof. Restrictiveness and forward correctability aim to protect high-level inputs. Guttman and Nadel [56] present a property, ND security, to secure both high-level input and output. The CSP definitions above make no distinction between types of event and aim to prevent interference by any high-level event.

The issue of composability which motivated the definitions in this section has continued to be explored by researchers. The CSP definitions such as *GC* have been shown [47] to be preserved by CSP parallel composition. Zakinthos and Lee [134, 135] suggested a technique to ensure that McCullough's "hook-up" composition will hold for generalised noninterference, *McC1*. This technique introduces delays into the system when feedback loops are present, achieving the same effect as the additional condition of the restrictiveness property. This approach makes the treatment of composition enforce security rather than trying to incorporate the hook-up property in the security definition. More general frameworks for the composition of security properties have been suggested, for example by Landauer and Redmond [76] and Dinolt *et al.* [32]. The former presented an abstract characterisation of module specification with properties defined as predicates joined by logical conjunction. In the latter, both components and policies were described as relations on "information units". Properties of closure and transitivity for such relations were used to provide conditions for combining components in ways which preserve security. McLean [88] classified information flow properties in terms of closure properties of certain trace-combining functions. This characterisation provides a basis for investigating the effect of various forms of composition. A similar direction was taken by Peri *et al.* [101] but with many- sorted predicate logic used as the framework. Recent work by Zakinthos and Lee [136] attempted to characterise security properties and

to derive a general theory for secure composition. This work was carried out in terms of trace sets and so applies only to deterministic systems.

2.3.5 Noninterference in other settings

McLean used a trace method [87, 85] for the abstract specification of software. A trace specification gives the names and types of all procedures (the syntax part) and a logical description of trace behaviour (the semantics). A set of axioms of the trace deductive system is provided, which supports formal verification of functional and security properties. The noninterference property requires that the output to a low-level user after any trace t is the same as after $t \upharpoonright H$. This can be applied directly to the trace specification, thus proving security at the most abstract level without the need for instantiation of state machine models that many of the properties have required.

Although trace-based definitions of noninterference are common, there are other approaches. Johnson and Thayer [69] used testing semantics of a labelled transition system. For each security level of the system a further labelled transition system is constructed which represents the possible states and actions at or below that security level. These are known as probes. A system is secure if it is indistinguishable (with respect to its probes) both before and after the occurrence of any high-level event. This provides a security property for nondeterministic systems with an interpretation going beyond the standard traces approach. It is closer to the failures/divergence model in CSP. Automata theory was used by Moskowitz and Costich [97] and an algebraic approach described by Pinsky [103]. Interpretations of noninterference for state- based notations include that of Bevier and Young [13]. Some of these approaches are reviewed with respect to the work of this thesis in Chapter 9.

2.4 Other approaches to limiting information flow

In addition to the noninterference approaches a number of other definitions for restricting information flow have been suggested.

2.4.1 Nondeducibility

Sutherland [127, 128] described the property of nondeducibility. Within the framework of nondeducibility it is sometimes possible for the actions of high-level users to affect lower level users without compromising system security. The important feature is whether the outputs that the lower level user sees actually convey any high-level information or not. Each user of a system has their own particular, possibly restricted, view of the system. If a user $u1$ can at any stage detect from their view of the system that certain possible views for another user $u2$ would be ruled out, then information is said to flow from $u2$ to $u1$. This is because $u1$ could deduce from their limited view that certain theoretically possible wider views would be incompatible, and hence infer information about what $u2$ sees. This definition relies on users knowing their own and others' interfaces to the system.

Nondeducibility can be expressed by observing how the views of the users are related. Suppose for any possible trace t of the system that $f1(t)$ gives $u1$'s view of the system and $f2(t)$ gives $u2$'s view. Define $f1 * f2$ by:

$$(f1 * f2)(t) = (f1(t), f2(t)) \qquad \text{for all } t$$

then information is said to flow between $u1$ and $u2$ iff $f1 * f2$ is not onto $(\text{ran}\, f1) \times (\text{ran}\, f2)$. This gives a symmetric interpretation of information flow. Sutherland showed that it is strictly weaker than the noninterference property *GM*. McCullough pointed out that nondeducibility is not preserved by hook-up composition.

2.4.2 Nondeducibility on strategies

A stronger version of nondeducibility was defined by Wittbold and Johnson [131]. This approach, known as nondeducibility on strategies, is intended to rule out cases where feedback of output to input can cause insecure information flow in a system for which basic nondeducibility holds. Wittbold and Johnson argued that it is not simply high- level inputs which are important, but also the strategy behind those inputs. Given this, a low-level view should be compatible not only with any high-level view, but also with any high-level strategy. The hook-up property does not hold for basic nondeducibility, but it does for nondeducibility on strategies as shown by Millen [91].

2.4.3 Universal theory of information flow

Foley [36] used CSP to state a universal theory of information flow in which information flow can occur from H to L when the actions of H can limit the possible traces for L. Much of Foley's work, for example [37, 38, 39, 40], has taken the wider view of describing ways of characterising information flow policies rather than concentrating on the definition of information flow. This can be seen as an extension of Denning's work on lattice- based information flow policies [28, 29] with additional security classifications representing objects of the system.

2.4.4 Separability

Another area which is similarly concerned with the use to which an information flow definition is put is that of separability. A separable system is one in which users can be isolated from one another. The idea of confining behaviour was introduced by Lampson [75] and the approach of separability was developed and applied by Rushby [112, 113, 114] who made use of a separation kernel. This maintains an environment in which each user is apparently separate, with communication allowed only through prescribed channels. CSP definitions have been given by Burnham [17], Jacob [64] and Graham-Cumming [47]. For example, Jacob's definition (the strongest of

these three) says that process P is separable if:

$$P = \|_{i \in I} Q_i$$

for some Q_i with disjoint alphabets. Roscoe and Wulf [111] investigated the relationship between separability and information flow, concluding that separability only ensures absence of information flow if each Q_i is deterministic.

2.4.5 Probabilistic interference

The properties referred to so far have been able to address a much wider range of information flows than previous methods, but they are still unable to deal with the issues of timing channels or probabilistic channels. To capture these it is necessary to extend the definitions and to model the system at a different level (for instance, the failures-divergence model of CSP might be replaced by a timed or probabilistic semantics). The problem of probabilistic channels where the probability of a low-level event occurring is influenced by high-level actions was addressed by Gray [50, 51]. The treatment of state machines is modified so that each transition is given an associated probability. If $p(s, e, s')$ is the probability that, starting from state s, the event e will move the system to state s', then the probability that, for user L, the system will move from state s to state s' is given by:

$$P_L(s, e, s') = \begin{cases} \sum_{t' =_L s'} p(s, e, t') & \text{if } e \in L \\ \sum_{f \notin L \wedge t' =_L s'} p(s, f, t') & \text{if } e \notin L \end{cases}$$

Gray's security definition extends restrictiveness by including the following requirement on probabilities:

$$s =_L t \Rightarrow P_L(s, e, s') = P_L(t, e, s')$$

Hence the probability that L sees a particular after-state must be the same from all L-equivalent starting states. This is referred to as P-restrictiveness. Probabilistic noninterference has also been defined by Gray [52].

2.4.6 Modal logic and causality

Several authors have used modal logic to express security properties. A modal logic allows statements about users' knowledge to be made directly by use of basic modal operators. For example, Glasgow *et al.* [42] used the logical formula:

$$K_{L\,\phi} \Rightarrow R_{L\,\phi}$$

to mean: if L knows ϕ then L is permitted to know ϕ. The R_L modal operator was also considered by Bieber and Cuppens [15] who viewed its semantics in a way which allows certain dependencies within a system. As with many of the previous definitions, a distinction is made between inputs, outputs and other events. The "secure dependencies" allow L to have knowledge of all values which are functionally dependent on L's input. With this view, a system is secure if for each trace[1] t:

$$K_{(t|L)\,\phi} \Rightarrow K_{(t|L_i)\,\phi}$$

where L_i is the inputs of L. This property, stated for traces, requires that L's view of a trace depend on L's inputs in the trace:

$$\forall\, t1, t2 : \mathit{Trace} \bullet t1 \upharpoonright L_i = t2 \upharpoonright L_i \Rightarrow t1 \upharpoonright L = t2 \upharpoonright L$$

The property is referred to as causality. The relationship between causality, nondeducibility, noninterference and generalised noninterference is explored by Bieber and Cuppens [15]. The latter three of these properties are all shown to be stronger than causality for deterministic systems partitioned between users H and L. The framework in which causality is defined allows only deterministic specifications. As discussed above, this limits both the systems that can be specified and the security analysis which is performed. Cuppens [27] defined the strictly weaker property, nondisclosure. A system which displays nondisclosure but not causality is regarded as "maybe secure". A final decision on security can only be made when more information is provided either through refinement or by providing probabilities.

[1] A trace in this system model is a function mapping each object at each time point to a suitable value.

Gray and Syverson [53] used a second order modal logic to reason about probabilistic systems. They show that causality is equivalent to probabilistic noninterference for deterministic, non-probabilistic systems.

2.5 Comment on noninterference properties

The information flow properties described in this chapter have proved to be theoretically very appealing and have provided the framework for much debate within the computer security research community. Perhaps part of the attraction is the wide variety of abstract properties which can be produced by small alterations in the definitions. It is then a challenging task to discover exactly which situations each definition covers and what systems each will pronounce secure. It is also important to compare the various definitions and to examine the differences between them. Until it is clear what each definition achieves and what it allows, there can be no rational way to decide between them. The development of "improved" security properties has unfolded in a "monster-barring" fashion [72] in which a proposed definition is found wanting by means of displaying an intuitively insecure system which the definition accepts as secure. A new definition is then constructed to deal with the problem case, only to be assailed by a fresh monster.

Another feature of the different security conditions is the number of different notations in which they are couched. For example, there are state machines [43, 44], labelled transition systems [68], CSP [3, 116], algebraic trace specifications [87] and abstract machines [14]. Although state-based notations have not generally been considered as suitable for expressing information flow properties, these too are represented [13]. A number of studies have made progress in the study of the definitions themselves, and in comparing the various approaches by casting them in a common notation. This is not a straight-forward task since features of the notation can often have a fundamental effect on the meaning of a definition. Attempts at comparison include that of Zakinthos and Lee [136] for trace properties; the categorical approach of [63]; [47] and [107] which use CSP and [35] using CCS. The

unifying notation for each of these latter three is a process algebra where an explicit treatment of state may not always be easy to achieve. On the other hand, properties expressed in a notation oriented towards capturing the state of a system may have limited scope for analysing the succession of events. When translating a definition from one notation to another for the sake of comparison it is necessary to show that the translation is faithful to the original. Otherwise the comparison may be misleading. For example, many of the definitions in this chapter make a distinction between inputs, outputs and other events. As shown by Bieber and Cuppens [15], even allowing for notation, strict comparisons may not be possible and account must be taken of different system assumptions such as totality of input. This great variety of notations continues to be an additional obfuscating aspect of computer security research. It is a barrier to scientific progress on top of the basic challenge of the underlying mathematics.

As a consequence of this diversity the developer of a secure system is faced with a bewildering array of definitions. There is little guidance as to how a choice can be made, what the implications of each decision are or how a chosen security definition may be incorporated into the development process as a whole. Perhaps this explains the poor take-up of the noninterference approach in industry. Guaspari *et al.* [54] referred to the proliferation of definitions as a "cottage industry". The project they described chose to use the Bell and LaPadula approach [11] with an *ad hoc* consideration of covert channels, rather than the more integrated approach of an information flow property. There is also the problem that a number of these information flow properties have intricate definitions including complicated trace expressions. These are not attractive to specify and can be difficult to verify. Part of the difficulty is that the condition is expressed as a property over all possible traces of a system. This can be overcome in some cases by the use of unwinding theorems. An unwinding theorem in general provides sufficient conditions which reduce the task of proving a property over all traces to the more tractable requirement of showing that each individual event maintains a certain property. Goguen and Meseguer [44] provided an unwinding theo-

rem for their original formulation of noninterference. Other examples of such theorems can be found in [116, 34, 92]. Even with an unwinding theorem, proof that a system satisfies a security property can be unwieldy. As pointed out by McLean [87]:

> Although noninterference is most naturally formulated as a trace specification, the traditional approach to proving a system noninterfering consists of constructing a finite state machine model of system operation, showing that the constructed machine satisfies a set of unwinding conditions that are sufficient for establishing noninterference, and mapping the state machine onto the software. Since the state machine constructed tends to embody, not an abstract specification, but a concrete mechanism for implementing that behaviour, the benefits of abstraction may be lost.

This is part of the motivation for McLean's trace specifications for which noninterference can be proved directly.

The very nature of security projects means that less information on them is made public than for general project developments, but for those examples for which some documentation is available it seems that methods pre-dating noninterference often prevail. These may well be better-known and viewed as "tried and trusted" but there is also the issue of ease of use. For example, Boswell [16] reported a recent project carried out by Logica, UK to specify a security policy for the NATO Air Command and Control System. The specification language used was Z and the policy was a combination of the Bell and LaPadula confidentiality model, the Clark and Wilson integrity policy [24] and two-person rule. Other examples are quoted in [122] and [54]. The subject of utility of security definitions is addressed further at the end of the next chapter.

Another point which should be addressed when considering the application of noninterference definitions concerns refinement. It might be hoped, and indeed expected, that if a security property is shown to be true of a system specification then that property will also hold for any implementation

of that specification. However, this is not true of most[2] of the definitions described in this chapter. It is an unfortunate feature that traditional refinement techniques do not preserve these security properties and that proving a specification secure against any of these properties is no guarantee that the final implementation will also be secure. This was noted by McCullough [81] and a general result for this area was set out by Jacob [66] whose "Basic Confidentiality Theorem" shows that functionality and confidentiality are, in a sense, opposites: the two properties are inversely related with an increase in one often leading to a decrease in the other. As an example of the so-called "Refinement Paradox" consider the Z operation AOp in Figure 2.1. It is intended here that *high*? is a value input by a user working at a high security classification and *low*! is an output to a user working at a lower classification. A summary of Z notation is given in Appendix C. AOp accepts a high-level input and nondeterministically outputs an unspecified low-level output. In this operation, the high-level operation does not interfere with the low-level operation. However, it can be refined by COp of Figure 2.1 which is intuitively insecure since high-level inputs are revealed as low-level outputs.

Refinement is a major problem for secure system development. It is obviously undesirable to expend effort verifying a security property of a specification only to have to re-verify the property for the implementation. A number of approaches (including ignoring the problem) have been taken. One method is to fix on a particular security property and to develop specialised refinement rules which *do* preserve it. This is the approach taken by Graham-Cumming [47] who developed rules for a CSP interpretation of noninterference. This provides a useful demonstration of the possibility of such an approach, but it also raises further difficulties. For a variety of different

[2]There are exceptions. These include the properties which apply only to deterministic systems, such as Goguen and Meseguer's original definition of noninterference and Bieber and Cuppen's causality. Another is McLean's trace specification approach [87] which allows only a limited amount of "harmless" nondeterminism. This can be prevented from causing any downward flow of information if the implementation is constrained always to resolve the nondeterminism in a uniform way. However, this could be viewed as placing extra requirements on the allowed refinements since only certain traditional refinements (namely, those which resolve the nondeterminism in a consistent fashion) are acceptable.

$$
\begin{array}{|l}
\hline
AOp \\
high? : \mathbb{N} \\
low! : \mathbb{N} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
COp \\
high? : \mathbb{N} \\
low! : \mathbb{N} \\
\hline
low! = high? \\
\hline
\end{array}
$$

Figure 2.1 Abstract and concrete operations in Z

security properties, a variety of different sets of refinement rules would be required. New tools would be needed for each, or at least, tools with different sets of rules. If this approach is taken then, for any variation of a security definition, we must completely rebuild the whole edifice of refinement theory.

Another, practical, response to the refinement paradox was produced by the ICL Secure Systems High Assurance Team. This group has used HOL [45] and Z on a range of security developments. Only systems which are functional and total may be considered secure. This rules out any further weakening of preconditions or resolution of nondeterminism through refinement. Again, these measures solve the problem, but are rather limiting. A specification can no longer be as abstract, with many minor details having to be decided at a high- level. The subject of abstractness of specification is addressed further at the end of Chapter 3.

The overall impression of information flow created by the definitions of noninterference brought together here is of a rather disorganised collection of competing properties. The implications of choosing one definition over another are not always clear; they can be hard to specify and expensive to verify. There is often little to guide the developer in incorporating a security property as part of a system development.

One question which arises is: what makes a good security property? It seems reasonable to wish to find a property which accords with intuitive ideas

of system security. For practical purposes there are other features which are also important, such as the ease of applying the property, the feasibility of verifying the property and the desirability of maintaining security through the development process. Perhaps at its most basic it might be hoped that such a property would be able to distinguish between a system in which information can pass between users and a system in which it cannot. The wide variety of requirements and interpretations means that this is much more demanding than it might at first appear. Different interpretations of what constitutes information and when it can be said to have been passed mean that an intuitive view of security for one person may be different from that of another. This prompts the question of whether there can be a single, basic definition of information flow which would also be flexible enough to adapt to the many different situations in which confidentiality is required.

The definitions of this chapter have each covered certain aspects of restricting information flow but none seems general enough to be regarded as an underlying definition in this way. In the next chapter a somewhat different property is considered which attempts to provide a more general, theoretically appealing definition of information flow, while at the same time addressing some of the practical considerations mentioned above.

2.6 Summary

This chapter described the various approaches which have been taken to define confidentiality. Those based on some form of information flow allow more general properties to be stated which can encompass both the obvious paths of communication and a number of more unexpected routes provided by covert channels. Despite the advantages of such definitions, there is little evidence of their use by industry or government. Chapter 2 suggested some reasons for this. These include the subtlety of the properties which allows many variations to arise, the difficulty of application and the barrier to development presented by the Refinement Paradox.

Chapter 3 reviews the use of determinism for security specification and

compares its utility with that of the properties in this chapter.

Chapter 3

Security through determinism

This chapter introduces a different approach to defining information flow by the limitation of nondeterminism in a system. The properties are described in some detail, with CSP examples provided to illustrate their use. The chapter concludes by examining what those definitions achieve and what further questions they raise.

3.1 The rôle of nondeterminism

The discussion at the end of Chapter 2 begins to reveal a feature common to the security definitions of that chapter: that is, the possibility that resolution of nondeterminism might introduce insecurities to the system. A system is nondeterministic in the CSP sense if it can make internal choices which affect a user's view of the system. Nondeterminism was exhibited by the schema AOp in Figure 2.1. The abstract operation allows a nondeterministic choice of output to the low-level user. The refinement to COp resolves this nondeterminism to constrain the value of the output. Unfortunately (from the point of view of security) the value assigned is that of the current high-level input. Thus, whilst nondeterminism towards a low-level user appears innocent enough, a malicious or unwary refinement could introduce insecurity.

A security definition which accepts AOp as secure must suppose that the internal choice is always resolved in a secure fashion. It is certainly true that there are many refinements of AOp which do not reveal the high-level input,

for instance:

$COp1$

$high? : \mathbb{N}$

$low! : \mathbb{N}$

$low! = 0$

Here, the low-level user sees only 0 each time the operation is executed, whatever the value of *high*?. However, using a traditional refinement path there is nothing to prevent *COp* from being taken as the chosen implementation with nondeterminism resolved by direct copying of high inputs to low outputs. It is also worth noting that, even with *AOp*, when the operation is executed a low-level user with knowledge of the system could infer that the high-level user had input *some* value, even if they were unable to tell what the value was. This might in itself be considered an undesirable flow of information.

The work of Roscoe and Woodcock [107, 110] made clear the link between determinism and security in a CSP framework. Roscoe gave noninterference definitions based on the limitation of nondeterminism within the system and demonstrated how these definitions in a sense underlie the earlier work on information flow. The security conditions proposed by Roscoe [107] were based on the limitation of nondeterminism within a system. The appearance of the system towards a low-level user is required to be completely deterministic. The way in which the "appearance to a low-level user" is defined is discussed below. If this view is deterministic then no information is said to flow from high to low. In the CSP definitions that follow we will assume that, unless explicitly stated otherwise, the alphabet of a process is partitioned into two sets: H consisting of high-level events and L consisting of low-level events. A low-level user can participate in L events only and should not be able to deduce anything about high-level activity. Keeping to a division of just two security levels simplifies the presentation, but the ideas can easily be generalised to accommodate the more general partial ordering of security levels commonly used.

3.1.1 Considering noninterference properties

The whole purpose of noninterference properties is to prevent (or at least, specify a limit to) high-level activity influencing what can be seen by a low-level user. This aim might be stated informally as:

> *whenever two traces of the system appear the same to a low-level user then the continued behaviour of the system must also be indistinguishable at the lower level.*

Roscoe [107] translated this into CSP terms and indicated the various choices of interpretation which might be made. Firstly, it is necessary to examine how "equivalence" is to be decided. In CSP, the various semantic models available (traces, failures- divergences, infinite traces etc.) give rise to various possible ways of assessing equality. Many of the security definitions from the previous chapter consider only traces equality and do not deal with the finer distinctions which can be made. A second choice concerns how the view at the lower level is determined. This consists of obscuring the high-level events in some way. One obvious way that this can be done in CSP is to use the hiding operator. If all H events are hidden, they become internal events of the process which will happen immediately and instantaneously with no external participation. This is certainly an effective way of obscuring H events, but the consequences of the way hiding is interpreted have implications for the information flow as demonstrated below.

With these possible choices the informal nondeterminism description given above might be rendered in CSP as:

$$\forall t, t' \in traces(P) \wedge t \upharpoonright L = t' \upharpoonright L \Rightarrow (P/t)\ obs\ H =_* (P/t')\ obs\ H$$

Where *obs H* could be hiding or some other method of obscuring H events and $=_*$ denotes equality which could be for traces or for failures-divergences. Different choices for these give rise to different properties which Roscoe [107] showed can characterise many of the security definitions of the previous chapter. For example, with hiding and trace equivalence this is basically equivalent to McCullough's generalised noninterference (property *McC1* from

Chapter 2). Other ways of observing the system and of abstracting high-level behaviour can lead to subtle differences in the definition and consequently in the systems classed as secure. With failures-divergences equivalence instead of trace equivalence, a stronger property will be obtained.

3.2 Security and determinism

The insight of [107] links information flow with nondeterminism. Another way to view the approach of the previous section is to note that if the appearance of the system to the low-level user L were completely deterministic then there would be no way that the behaviour of a high-level user could possibly influence L's view. Nothing at all can influence a completely determined view. L's view of the system is its interface with the system. Hence to ensure that the system is secure with respect to the actions of H we can define L's interface with the system and require it to be deterministic. As mentioned in the discussion of the previous section, there are several ways in which we can abstract away H events to give L's interface. This is reflected in the properties defined below.

3.2.1 Eager deterministic security

The first security definition based on determinism obscures H events by hiding them.

Definition 1 *Eager Deterministic Security (Roscoe) A process P is eagerly secure with respect to H iff*

$$P \setminus H \quad \textit{is deterministic}$$

The term "eager" comes from the interpretation of hidden events which views them as occurring internally whenever and as soon as they can. They occur instantaneously and cannot delay the system. The definition uses the standard CSP interpretation of determinism in which a process P is deterministic if:

$$\mathit{divs}(P) = \emptyset \wedge (tr^\frown\langle x\rangle \in \mathit{traces}(P) \Rightarrow (tr, \{x\}) \notin \mathit{fails}(P))$$

This requires that there is no point at which an event could be offered by the system but might also be refused. According to Definition 1, all H events must first be hidden in process P, making them internal events which happen as and when they can. Any influence that H might have on L will now appear to L to be the result of arbitrary internal decisions, that is, as nondeterminism in $P \setminus H$. Conversely, if $P \setminus H$ is deterministic, then no behaviour of H can be capable of influencing L's view and hence P is secure. To see how the definition is used, consider the following example.

Example 2 With $H = \{h1, h2\}$ and $L = \{l1, l2\}$:

$$P1 = (h1 \rightarrow l1 \rightarrow P1) \,\Box\, (h2 \rightarrow l2 \rightarrow P1)$$

$P1$ offers an initial external choice between $h1$ and $h2$. The event offered to L is thus dependent upon H's choice. To assess $P1$ with respect to Definition 1 H is hidden within the process:

$$P1 \setminus H = (l1 \rightarrow (P1 \setminus H)) \sqcap (l2 \rightarrow (P1 \setminus H))$$

$P1 \setminus H$ makes a nondeterministic choice between $l1$ and $l2$ and therefore does not satisfy Definition 1. The process $P1$ is insecure because the choice of low level action depends directly on the behaviour of H. This is revealed by the nondeterminism of $P \setminus H$. ♄

This approach is referred to as *eager* because of the way in which hidden events are assumed to occur internally whenever and as soon as they can. There will be no observable delay to L (as long as the system does not diverge). This can make a difference to the assessment of h's influence as shown in the next example.

Example 3 Let $H = \{h\}$ and $L = \{l\}$:

$$P2 = h \rightarrow l \rightarrow P2$$

then hiding H actions gives simply :

$$P2 \setminus H = l \rightarrow (P2 \setminus H)$$

which is deterministic. ♄

$P2$ is considered secure according to Definition 1 even though occurrences of l depend entirely on h having taken place. The hiding process conceals this: it is as if h happens so quickly that no refusal could be observed by L.

3.2.2 Lazy deterministic security

To rule out dependencies such as that in $P2$ above, Roscoe [107] gave a second deterministic security definition in which high-level events are obscured not by hiding but by allowing them to occur at any point. This method of obscuring H events is rather less obvious since it does not conceal the high-level events but abstracts from their specified order of occurrence within the process. This form of obscuring can be achieved by interleaving the original process with RUN_H where:

$$RUN_H = h : H \rightarrow RUN_H$$

which is always prepared to offer any H action. The second deterministic security definition obscures H by interleaving with RUN_H.

Definition 2 *Lazy Deterministic Security (Roscoe) A process P is lazily secure with respect to H iff*

$$P \,|||\, RUN_H \quad \textit{is deterministic}$$

This is referred to as "lazy" abstraction. A low-level user cannot know whether H events are contributed by P or by RUN_H. Here, H events are obscured since they may be contributed either by the original process P or by RUN_H. The following examples show how dependencies are revealed by Definition 2.

Example 4 Considering the process $P1$ from Example 2:

$$P1 \,|||\, RUN_H = ((h1 \rightarrow l1 \rightarrow P1) \,\Box\, (h2 \rightarrow l2 \rightarrow P1)) \,|||\, RUN_H$$

One possible trace of $P1 \,|||\, RUN_H$ is $\langle h1, l1 \rangle$. However, $(\langle h1 \rangle, \{l1\})$ is also a failure, and so $P1 \,|||\, RUN_H$ is nondeterministic. Thus $P1$ does not satisfy Definition 2. ♮

Example 5 Considering the process $P2$ from Example 3:

$$P2 \,|||\, RUN_H = (h \rightarrow l \rightarrow P2) \,|||\, RUN_H$$

Again, $\langle h, l \rangle$ is a possible trace and $(\langle h \rangle, \{l\})$ is a failure. So $P2 \,|||\, RUN_H$ is nondeterministic and $P2$, which did satisfy the condition for eager security, is not lazily secure. The dependency of l on h is revealed by this method of obscuring H events. ♮

3.2.3 Mixed deterministic security

The definitions of eager and lazy security amount to the same thing in many cases, but the method of abstraction does have important differences which will be made clear in the examples of the next section. In certain circumstances, it may be useful to be able to apply the eager condition to part of the system and the lazy condition to the rest. This can be used for situations where some high-level events (known as signal events) are viewed as happening so quickly that no delay can be observed by L, whilst the remaining high-level events (known as delay events) could cause an observable refusal to L. Again, the use of this is illustrated in the examples of the next section.

Definition 3 *Mixed Deterministic Security (Roscoe) Suppose system P has high-level actions $H = D \cup S$ where S is the set of signal events and $D \cap S = \emptyset$. Then P satisfies the mixed security condition iff:*

$$(P \setminus S) \,|||\, RUN_D \text{ is deterministic}$$

The deterministic security properties will give similar results to previous security properties for a wide range of systems, but there are important differences. The deterministic definitions go beyond the weaker test of trace equivalence discussed in Section 3.1.1, since the definition of determinism requires that failures and divergences be taken into account. However, the requirement that all nondeterminism at the interface of the system with the low-level user be banned goes further even than a definition in the style of those in Section 3.1.1 with failures-divergences equivalence. These points are illustrated in the examples of the following section.

3.2.4 Examples

Use of the three definitions and the differences between them are shown in the following examples. Though the deterministic security properties are precisely and concisely stated, their full implications are perhaps not immediately apparent from the definitions. A range of examples can highlight some of the important points. This continues on from Examples 2 to 5 above which show the basic approach to applying the definitions and the difference between eager and lazy security. In the processes used below it is assumed that H is the set of events used in the process definition whose names start with h, and similarly for L and l.

Example 6 This example shows a process which is both eagerly and lazily secure.

$$Q1 = (h \rightarrow l \rightarrow Q1) \,\Box\, (l \rightarrow Q1)$$

$Q1$ always offers the choice of l no matter what the activity of H. Hiding H gives:

$$Q1 \setminus H = l \rightarrow (Q1 \setminus H)$$

which is deterministic since it always offers l. Hence the process $Q1$ satisfies the eager deterministic security property with respect to H. $Q1$ can also be tested for lazy deterministic security. Interleaving with RUN_H always gives a process which can never refuse any H events. Also, since $Q1$ is always prepared to offer l, $Q1 \;|||\; RUN_H$ cannot refuse l either. The system does not diverge and cannot refuse any event in its alphabet and is therefore deterministic. Hence $Q1$ is also lazily secure. ♮

Example 7 This example again illustrates the difference between the two basic definitions. Here, L's event is offered in both branches whatever the choice of H event has been.

$$Q2 = (h1 \rightarrow l \rightarrow Q2) \,\Box\, (h2 \rightarrow l \rightarrow Q2)$$

Hiding H actions gives simply :

$$Q2 \setminus H = l \rightarrow (Q2 \setminus H)$$

which is deterministic. $Q2$ is considered secure according to Definition 1 even though occurrences of l depend entirely on an H event having taken place. It is as if $h1$ and $h2$ happen so quickly that no refusal could be observed by L. In comparison, the lazy approach takes the interleaving:

$$\begin{aligned} & Q2 \,|||\, RUN_H \\ & = ((h1 \rightarrow l \rightarrow Q2) \,\Box\, (h2 \rightarrow l \rightarrow Q2)) \,|||\, (x : H \rightarrow RUN_H) \end{aligned}$$

Here, the event $h1$ is possible for both operands. So, for example, $\langle h1, l \rangle$ is a possible trace, but $(\langle h1 \rangle, \{l\})$ is also a failure. Hence the process is nondeterministic and so $Q2$ is considered insecure according to Definition 2. In this case, the dependence of l on h is significant and is deemed to compromise the security of the process.

The choice between eager and lazy definitions depends largely on how the events of the system are viewed. In general, the lazy definition is probably more useful since dependencies as in $Q2$ are ruled out (but see also Example 9). ♮

Example 8 The determinism definitions do not allow nondeterminism at the interface of the system with the low-level user, but they do not require the overall system to be deterministic.

$$Q3 = l \rightarrow Q3 \,\Box\, ((h1 \rightarrow l \rightarrow Q3) \,\sqcap\, (h2 \rightarrow l \rightarrow Q3))$$

This process allows a high-level internal choice between events $h1$ and $h2$. However, the eager deterministic security condition hides both high-level events to give a process which always offers l and so is deterministic. Also for the lazy property, interleaving $Q3$ with RUN_H gives a process which never refuses any event of its alphabet and is deterministic. Hence $Q3$ is secure by both definitions. ♮

Example 9 It is not the case that lazy security can simply be said to be stronger than eager security: there are cases where a process is lazily secure but not eagerly secure.

$$Q4 = h \rightarrow Q4 \,\Box\, l \rightarrow Q4$$

Hiding h in $Q4$ causes divergence, so $Q4 \setminus H$ is not deterministic and hence $Q4$ fails to meet the requirement for eager security. But interleaving with h gives a different result. $Q4$ can produce any trace with elements from $\{h, l\}$ and has no refusals. It is also divergence-free. Interleaving with RUN_H does not alter the system ($Q4$ is always prepared to offer h, so nothing new is added). So $Q4 \,|||\, RUN_H$ is deterministic and the system is lazily secure. ♮

The failure of $Q4$ with respect to Definition 1 comes about because of the divergence introduced by hiding. It is in theory possible to have an infinite sequence of h. When hidden, this infinite sequence results in a process which will have no further external communication and L is permanently blocked. Whether this really is worth worrying about again depends on the interpretation of events. If an implementation of $Q4$ could ensure that the choice between h and l allowed l to gain access (for example, by giving priority to l in some way) then the possibility of H constantly monopolizing the system would not arise. For this reason, lazy security is generally sufficient and a system such as $Q4$ would be regarded as secure. The combination of eager and lazy security was referred to by Roscoe as "strong security". It is worth noting that when $P \setminus H$ is divergence-free then lazy security *does* imply eager security, as shown by Roscoe [107].

Example 10 Whilst L events are not allowed to depend on H events, high-level events may be influenced by low-level ones. For instance, compare:

$$Q5 = h \rightarrow (l \rightarrow Q5 \,\Box\, h \rightarrow Q5)$$
$$Q6 = l \rightarrow (l \rightarrow Q6 \,\Box\, h \rightarrow Q6)$$

With $Q5$, hiding H causes divergence. Taking the interleaving $Q5 \,|||\, RUN_H$, $\langle h, l\rangle$ is a possible trace, but $(\langle h\rangle, \{l\})$ is also a failure. So $Q5$ fails both

conditions. On the other hand:

$$(Q6 \setminus H) = l \rightarrow (Q6 \setminus H)$$

and so Definition 1 is satisfied. Also,

$$Q6 \,|||\, RUN_H = (l \rightarrow (l \rightarrow Q6 \,\Box\, h \rightarrow Q6)) \,|||\, (x : H \rightarrow RUN_H)$$

This can never refuse any H or L event and is deterministic. Hence Definition 2 holds too. ♮

In many cases, the determinism definitions are equivalent to the noninterference definitions outlined in Section 3.1.1. The following examples reveal some of the differences.

Example 11 A definition which tests for equality of traces cannot deal adequately with nondeterministic systems such as:

$$Q7 = l \rightarrow (l \rightarrow Q7 \sqcap h \rightarrow Q7)$$

If the low-level user observes a refusal then it must be because the system has chosen to offer h and not l. Event l only becomes available again once h has occurred. Lazy security fails to hold for $Q7$ since, for instance, $\langle l, l \rangle$ is a trace of $Q7 \,|||\, RUN_H$ and $(\langle l \rangle, \{l\})$ is also a failure. However, considering the trace-equivalence interleaving property it is necessary to check that:

$$t, t' \in traces(P) \wedge t \upharpoonright L = t' \upharpoonright L \Rightarrow (P/t)\ obs\ H =_* (P/t')\ obs\ H$$

for $Q7$ with $=_T$ and $|||\, RUN_H$. This is true since $Q7 \,|||\, RUN_H$ has every trace of sequences of $\{l, h\}$. ♮

For a deterministic process the deterministic properties are equivalent to their counterparts in the noninterference definitions of Section 3.1.1. However, for a nondeterministic process, neither the trace-equivalence properties nor the failures-divergences equivalence properties can be relied upon.

Example 12 The noninterference definitions which test for equality of failures-divergences do distinguish between deterministic and nondeterministic processes, but there are still important differences. The determinism definitions rule out the possibility of any nondeterminism at the interface with L, whereas all the conditions characterised in Section 3.1.1 permit some nondeterminism. Consider the process $HAVOC_{\{h,l\}}$[1] where:

$$HAVOC_H = STOP \sqcap (x : H \rightarrow HAVOC_H)$$

This process never diverges but it can at any point refuse every event in its alphabet. By checking the inclusion of failures and divergences, the insecure process $Q1$, for example, can be shown to refine $HAVOC$:

$$HAVOC_{(H \cup L)} \sqsubseteq Q1$$

From the point of view of lazy deterministic security, $HAVOC_{(H \cup L)}$ is insecure since it is nondeterministic towards L. However, since for all traces t:

$$HAVOC_{(H \cup L)}/t = HAVOC_{(H \cup L)}$$

the noninterference conditions from Section 3.1.1 all accept it as secure. Any definition of security which accepts $HAVOC$ as secure immediately gives rise to the Refinement Paradox. This example illustrates a very important feature of the deterministic security properties. The implications of this treatment of nondeterminism are discussed in Section 3.3. ♮

Example 13 The fact that the nondeterminism conditions represent very stringent conditions is further illustrated by the following process in which there is no high-level activity at all.

$$Q8 = (l1 \rightarrow Q8) \;|||\; (l2 \rightarrow Q8)$$

Both hiding and interleaving with the empty set of H events simply gives $Q8$ but this is itself nondeterministic. So $Q8$ is insecure by both deterministic definitions. ♮

[1]This is equivalent to the process referred to by Roscoe as $CHAOS_H$. It is given a different name here to avoid any possible confusion with the different definition of $CHAOS$ given by Hoare [59].

Example 14 The mixed security condition can be used when some high-level events, such as outputs, cannot be resisted by the high-level user and would not make refusals apparent at the lower level. For instance:

$$Q9 = h1?x \rightarrow h2!f(x) \rightarrow Q9$$
$$\Box\ l \rightarrow Q9$$

For any value v, $\langle h1?v, h2!f(v), l\rangle$ is a possible trace of $Q9 \ |||\ RUN_H$ and $(\langle h1?v, h2!f(v)\rangle, \{l\})$ is a failure. So $Q9$ fails to meet the lazy deterministic condition. However, the output event $h2$ could be given an eager interpretation, since the high-level user cannot choose to delay the system by not accepting the output $h2$. For the security analysis of this process we apply the mixed security condition with $D = \{h1\}$ and $S = \{h2\}$, which amounts to checking that:

$$(Q9 \setminus \{h2\})\ |||\ RUN_{\{h1\}}\ det$$

This is always prepared to offer both $h1$ and l, and is therefore deterministic. So, under the mixed security condition, $Q9$ is considered secure. ♮

3.3 An alternative formulation of deterministic security properties

The mixed security definition of Example 14 is perhaps the first step towards recognising the needs of a real system: output events at the high-level would not in reality be viewed as delaying the system, so they should be accommodated by a view of security which takes this into account. Roscoe [107] introduced an alternative way to express the three conditions above. The approach is important because it paves the way for constructing a wider variety of security policies which may be more generally applicable.

3.3.1 Specifying abstract high-level behaviour

The alternative way to specify deterministic security conditions is to construct a process U which represents the high-level behaviours for which se-

curity should be guaranteed. The condition to be checked to show security of P is:

$$(P \underset{H}{\|} U) \setminus H \text{ is deterministic}$$

Using this construction, Roscoe [107] showed that Definitions 1, 2 and 3 are equivalent to definitions in a similar format:

Eager deterministic security

$$(P \underset{H}{\|} RUN_H) \setminus H \text{ is deterministic}$$

Lazy deterministic security

$$(P \underset{H}{\|} FINITE_H) \setminus H \text{ is deterministic}$$

Mixed deterministic security

$$(P \underset{(D \cup S)}{\|} (RUN_S \;|||\; FINITE_D)) \setminus (D \cup S) \text{ is deterministic}$$

Strong deterministic security

$$(P \underset{H}{\|} HAVOC_H) \setminus H \text{ is deterministic}$$

$FINITE_X$ has the same failures-divergences as $HAVOC_X$ but has no infinite traces (see Appendix A). This approach specifies the range of high-level activity within which the system is required to remain secure. Constructing different processes for U allows different policies to be specified.

3.3.2 Conditional security

One important relaxation of the basic noninterference condition noted by its original proponents [43] is the freedom to specify conditional security policies. A conditional property modifies the noninterference requirement by the addition of a further condition upon which the security of the system is dependent. That is, whenever the condition holds, the system is required to obey the noninterference property. Specifying the most abstract behaviour

of the high-level user under which the system is expected to remain secure represents a move towards the possibility of expressing conditional security properties.

Example 15 A system manager with high-level actions H is given the discretion to perform certain managerial tasks, $M \subseteq H$, when judged necessary. The conditional noninterference condition for this is:

$$(P \Vert_{H} FINITE_{(H-M)}) \setminus H \text{ is deterministic}$$

$\natural$

3.4 Considerations for a secure development method

The deterministic security properties have several important advantages over previous approaches. From a practical point of view they are easy to state and their intention is clear. This is a major improvement over definitions involving obscure restrictions on perturbations of traces. The determinism properties have the appealing feature of being maintained through refinement. This is a result of the fact that deterministic processes are maximal under refinement. It makes a development considerably easier if a security property can be verified at an abstract level and then needs no further checking. Traditional refinement rules and tools based on them can be used with confidence with no hidden security implications.

Another useful aspect of the determinism conditions is the way in which they can be verified automatically. The CSP model-checker, FDR (abbreviating Failures Divergence Refinement) [105], has been used for this purpose[2]. The possibility of automatic verification such as this is a strong point in favour of definitions based on determinism and greatly increases their utility.

[2]FDR checks behavioural properties of CSP specifications and has been used to verify process refinement. Although determinism is a somewhat different property from the ones for which FDR was originally designed, the model- checker's capabilities can be harnessed to verify the determinism-based security properties as described by Roscoe [107]

The theoretical basis of determinism properties is also very appealing. Roscoe [107] discussed at length the nature of nondeterminism, both probabilistic and as the result of under-specification. A fully-defined probabilistic specification describes an event which is required to appear genuinely random and can state the required distribution of the nondeterministic event. (However, it appears that care might still be needed since a random low-level output stream with 0s and 1s equally likely could be implemented by copying the values of a similarly random high-level component, such as a key stream. The distributions might be the same, but the system should not be regarded as secure.) Under-specification occurs when the specifier does not care which of a range of possible options is chosen. As illustrated in Figure 2.1, the danger comes with the possibility of an implementation resolving the nondeterminism in an insecure way. Banning nondeterminism completely from the low-level user's interface ensures that no such insecure information flow can be introduced.

Another advantage of the determinism conditions is that they achieve a greater degree of model-independence than many others. The notion of nondeterminism is unchanged across the different semantic models of CSP (apart from the timed ones). It is also equivalent for many other concurrent notations which can discriminate beyond the traces of a system. So the same security definitions can be used in each case. It is obviously not possible for security flows to be eliminated in a notation which does not have the ability to analyse the situation. Just as a trace-based notation cannot deal with the issue of nondeterminism, so timing channels cannot be considered in an untimed model.

When assessing the utility of a security definition it is necessary to consider both its suitability for the theoretical modelling of security properties and how it can be fitted into the development process of a secure system. An event calculus is able to deal directly with sequences of events and to make clear the interactions between them. However, it is also true that for general specification purposes a state-based notation such as Z [125] can be very useful. This style facilitates the functional description of a system,

supporting abstract modular specification and allowing refinement of both operations and data structures. The appeal of such an approach is reflected in the popularity of state-based notations. Z in particular has been used in the development of many secure systems and the use of formal methods is mandated by various government standards as described by Sinclair and Ince [122]. However, the attempts have often been unwieldy and many note a number of difficulties, such as the inadequacy of the notation to express event interaction, the effects of refining nondeterminism and the way in which weakening preconditions in refinement.can affect security.

The utility of such a state-based approach to functional specification was acknowledged by Roscoe *et al.* [110] where a development consisting first of a state-based specification followed by analysis of a process algebra representation of the system is proposed. The example they gave uses three levels of specification of a secure file system, starting with Z, then representing the Z as an action system and finally translating the action system into CSP. FDR can then be used to check the deterministic security conditions. This route demonstrates the feasibility of this kind of development, using existing correspondences between the languages employed. However, it also shows some of the disadvantages inherent in any development method involving wholesale translations between notations.

Firstly, translations themselves can introduce as many problems as they solve. Representing one notation within another involves making a number of decisions about interpretation. In the case of action systems, the CSP failures-divergences semantics given by Morgan [93] put this on a firm footing. However, the "informal but systematic" translation from Z to action systems used in [110] begs the question of how appropriate informal interpretations are for secure developments. In this case, a Z specification which is nondeterministic in its output to a low-level user is translated to a deterministic, secure action system by the particular interpretation that is placed upon it. The *Read* option for the file system is constructed from a successful case in which file data are output, and various error cases. If an unknown file identifier is entered, the values of the output data are not specified. Since

this, in theory, allows any implementation the freedom to choose any data for this purpose, the specification could obviously be refined in an insecure way by the output of data from high-level files. This is an example of the Refinement Paradox. The CSP interpretation chooses not to output any valid data in such circumstances. This may seem a very sensible choice, but it leads to the situation where a Z specification with possible insecure refinements is declared secure by its CSP translation.

Secondly, each translation introduces an extra step during which errors may appear. Certainly, anything other than a completely automated process increases the opportunity for human error (and even an automated process could suffer from human error). This is particularly unwelcome in the development of secure systems. A proliferation of notations is also a deterrent for those engaged in practical developments who must become proficient in a number of notations and who may well require tool support for each stage of the process.

A further consideration with a translation approach is how errors discovered in the process algebra should be related back to the original specification. It may not be clear how a particular insecurity can be isolated and removed in the original version. Simply altering the final-stage specification may result in the specifications becoming inconsistent; this provides a poor record of the development and makes maintainance at a later date more difficult.

These points suggest that it would be desirable to find a method which combines the advantages of both state-based and event-based notations, while avoiding the need for translation of notations. One approach which is investigated in the following chapters is to use action systems for both specification and proof of security properties. The CSP interpretation of Morgan [93] allows criteria such as Definitions 1, 2 and 3 to be considered directly for action systems rather than translating the action system to CSP. Work done by Butler [20, 18] on the refinement of action systems suggested that this is a promising line of attack, applicable to systems of some complexity. Although it is possible to use an algebraic style of CSP (as in the final specification in the example of Roscoe *et al.* [110]), an action systems approach allows

a much more direct consideration of state. It is also a straightforward procedure to refine action systems using the method of simulation. This treats both state and events in a single unified step and provides a natural route for the integrated development of both of these aspects of the secure system.

Another advantage of the combined state and events approach is that it can provide the means for expressing and analysing more complex security policies. In practice, it is rarely the case that noninterference conditions alone are required. There is usually an interconnected collection of requirements as in [99], or a combination of recognised components such as used in the project reported by Boswell [16] which demands multi-level security, integrity and two-person rule. Although Roscoe *et al.* [110] made the case for not having a separate security policy (no need for tedious verification that it is upheld) in more complicated situations an explicitly stated security requirement may be a necessity.

Even from the viewpoint of multi-level security it may not be possible to investigate fully the security of a system by considering sequences of events alone. There may be a need to know what is communicated and to consider the classification of information as well as of actions. For example, the process:

$$T = h \rightarrow T \,\Box\, l \rightarrow T$$

is lazily secure according to Definition 2. Yet if h is an event which updates array a with a high-level message and l is an event which can display the contents of a to a low-level user, this would hardly be considered secure. Obviously this is over-simplified: if an output is concerned then the CSP representation should reflect this and the dependency revealed. However, it makes the point that interpretation of events may also be important. This issue is also raised in Chapter 8 where a case study is presented which makes use of both secure events and secure state.

The determinism conditions, whilst theoretically appealing and simple to apply, are very rigid in their requirements. As well as needing to combine noninterference with other policies, it is unrealistic to expect that a strict noninterference policy will be sufficient for real systems. If the determinis-

tic security definitions are viewed as a sound theoretical basis for analysis of security, it remains to provide a means of adapting these definitions to the realities of actual systems and incorporating them into the development process.

One last comment on desirable attributes of security developments concerns the value of abstraction. The major benefit of specification has often been said to be the way in which it enables us to abstract away from implementation details and to concentrate on different concerns at appropriate levels. To demand a top-level specification which is fully secure in the sense of the definitions of Section 3.1.1 places limitations on a specifier's freedom of abstraction. There is a significant difference between a specification displaying "don't care" nondeterminism which an unscrupulous implementor could exploit, and a specification about which we "don't care at this level". Some specifications are definitely insecure. A specification in which high-level inputs are written out to a low-level user would come into this category. Others, like the Z example of Roscoe *et al.* [110], have no actual insecurity at the current level but could be refined to either a secure or an insecure implementation. It seems that this may well be a distinction worth noting.

3.5 Summary

This chapter and the previous one have outlined existing approaches to the formal statement of security properties and to the development of secure systems. The properties described have each aimed to provide a formal description of what it means for a system to be secure, and to enable the systematic verification of the property for individual systems. The definitions based on controlling information flow within the system have made a significant contribution to the understanding of the concept of confidentiality and have encompassed wider issues of system security, such as covert channels, which simple access controls cannot address.

Despite the progress made in this area, the proliferation of similar definitions with subtle differences indicates a continued debate over what con-

stitutes an appropriate definition. It has become difficult to appreciate the significance of the distinctions between definitions and practitioners have found it hard to choose between them.

Chapter 3 has concentrated on one particular approach which provides a convincing theoretical basis for confidentiality. In particular, it provides an explanation for why refinement of secure systems can be problematic. However, there has so far been little experience of applying this work in the context of general systems' development. The results as they stand are not readily applied to the formal development languages most commonly used in practice. As discussed above, the developers of real systems often require the flexibility afforded by allowing direct manipulation of state within the development process and of unifying the specification of both security and functional properties. These are the areas not covered by existing work which this thesis aims to address.

The next chapter introduces action systems as a suitable notation for investigating such an integrated approach. It is unreasonable to expect that their use can accommodate all possible features, yet it provides a useful starting point for examining the use of an approach which gives equal consideration to the state and events of a system.

Chapter 4

Action systems and nondeterminism

One of the desirable features of security developments identified in the previous chapter was the ability to combine an analysis of events with the specification of state, and to be able to refine both aspects in a uniform manner. This chapter investigates the use of action systems for this purpose. The chapter starts with an introduction to action systems. This includes the correspondence of action systems with CSP for failures-divergences and infinite traces and covers both value-passing action systems and those with internal actions. This level of analysis is required for the definitions of determinism and system security which are derived later in the chapter. Examples of action systems are given throughout the chapter.

4.1 Introduction to action systems

Action systems originate from the work of Back and Kurki-Suonio [7] and provide a state-based approach to the specification of concurrent systems. Informally, in an action system a collection of labelled actions share a state. An enabled action is selected and executed, resulting in a possibly altered state. Again, an action is selected from those currently enabled and so execution of the action system proceeds. There has been a good deal of work developing the use of action systems, such as that of Back *et al.* [6, 8, 10]. The work of this chapter follows the approach of Morgan [93] and Butler [20] by using the semantic model provided by the correspondence of action systems with

CSP. This approach has several advantages, both in terms of inheritance of CSP properties and in the way that refinement is viewed. Two particularly useful aspects are that, firstly, refinement in the CSP approach distinguishes between internal and external nondeterminism (that is, between choice resolved by the environment and choices made internally) and, secondly, action systems may be refined into parallel components which may themselves be further decomposed without constraints being placed on their environment.

4.1.1 Actions

An action may be thought of as a guarded command. The guard is a predicate, and an action is said to be enabled in any state in which its guard is true. The command is a program fragment from Dijkstra's guarded command language [31] in which a command is defined in terms of its weakest precondition. The weakest precondition for command c to attain postcondition α is the set of all states from which execution of c is guaranteed to terminate in a state satisfying α. It is denoted:

$$wp(c, \alpha)$$

Appendix B gives Dijkstra's wp definitions for the program statements of the guarded command language used here.

A related concept which proves very useful is the conjugate weakest precondition of a command, $\overline{wp}$. This is the set of states from which execution of the command might possibly establish a given postcondition:

Definition 4 *(Morgan) For command c and postcondition α:*

$$\overline{wp}(c, \alpha) \mathrel{\widehat{=}} \neg\, wp(c, \neg\, \alpha)$$

For the correspondence between CSP and action systems Morgan [93] extended Dijkstra's guarded command language by giving a meaning to "naked guarded commands" of the form:

$$g \rightarrow c$$

where the predicate g is the guard of command c. The weakest precondition is defined:

Definition 5 *(Morgan) For guard g, command c and postcondition α:*

$$wp(g \rightarrow c, \alpha) \mathrel{\hat{=}} g \Rightarrow wp(c, \alpha)$$

It follows that:

$$\overline{wp}(g \rightarrow c, \alpha) \mathrel{\hat{=}} g \wedge \overline{wp}(c, \alpha)$$

The weakest preconditions of guarded commands do not obey all the laws which obtain for the basic commands of Dijkstra's language. In particular, the Law of the Excluded Miracle, which holds for each command, c of Dijkstra's language:

$$wp(c, \mathit{false}) = \mathit{false}$$

does not hold for some guarded commands. A guarded command $g \rightarrow c$ is said to be *miraculous* if it fails to obey this law. For example:

$$wp(\mathit{false} \rightarrow \mathit{skip}, \mathit{false})$$
$$\equiv \mathit{false} \Rightarrow wp(\mathit{skip}, \mathit{false})$$
$$\equiv \mathit{true}$$

One useful property is the distributivity of disjunction for conjugate weakest precondition of guarded commands which follows from the distributivity of conjunction for basic commands:

Property 1 *$\vee$-distribution*

$$\overline{wp}(g \rightarrow c, \alpha \vee \beta) \equiv \overline{wp}(g \rightarrow c, \alpha) \vee \overline{wp}(g \rightarrow c, \beta)$$

Proof

$$\overline{wp}(g \rightarrow c, \alpha \vee \beta)$$
$$\equiv g \wedge \overline{wp}(c, \alpha \vee \beta) \qquad \text{[Defn. } \overline{wp}\text{]}$$
$$\equiv g \wedge \neg\, wp(c, \neg\, \alpha \wedge \neg\, \beta)$$
$$\equiv g \wedge \neg\, (wp(c, \neg\, \alpha) \wedge wp(c, \neg\, \beta)) \qquad [\wedge\text{-dist for } c]$$
$$\equiv g \wedge (\neg\, wp(c, \neg\, \alpha) \vee \neg\, wp(c, \neg\, \beta))$$
$$\equiv g \wedge (\overline{wp}(c, \alpha) \vee \overline{wp}(c, \beta))$$
$$\equiv (g \wedge \overline{wp}(c, \alpha)) \vee (g \wedge \overline{wp}(c, \beta))$$
$$\equiv \overline{wp}(g \rightarrow c, \alpha) \vee \overline{wp}(g \rightarrow c, \beta)$$

Although an action is defined here as a guarded command, it may be represented in other ways provided a weakest precondition semantics can be given. For instance, Butler [20] used specification statements [95] as a convenient notation for specifying actions. Details of specification statements and their weakest preconditions are to be found in Appendix B. There is no reason why other suitable notations, such as Z, should not also be appropriate. A weakest precondition semantics for Z has been given by Cavalcanti and Woodcock [23] which paves the way for the formal incorporation of Z representing actions in action systems. For an action a specified without an explicit guard, the guard, gd, can be calculated using:

$$gd\ a = \overline{wp}(a, true)$$

This is the set of states in which a is enabled. It is easy to show that this gives the expected result if applied to a guarded command since:

$$\begin{aligned}
&\overline{wp}(g \rightarrow c, true) \\
&\equiv g \wedge \overline{wp}(c, true) && \text{[Defn. of } \overline{wp}\text{]} \\
&\equiv g \wedge true && [c \text{ is a command}] \\
&\equiv g
\end{aligned}$$

4.1.2 Action systems

An action system $\mathcal{A} = (A, S, \mathcal{A}_{init}, \mathcal{A}_A)$ consists of a state S, an initialisation $\mathcal{A}_{init}$, and a set of labelled actions $\mathcal{A}_A$ where A is an alphabet of labels. Figure 4.1 shows the way in which such an action system is written. The state, S, consists of a variable or variables which may be related by a state invariant. If an invariant is present it is taken to be part of the precondition and postcondition for each action. The state variables are common to all actions of the system. The initialisation is a command (although, again, this could be represented by a Z schema or a specification statement). For an action system to be well-formed, the initialisation must always be enabled

$$\mathcal{A} \triangleq \left(\begin{array}{l} \textbf{var } S \\ \textbf{initially } \mathcal{A}_{init} \\ \vdots \\ \textbf{action } a \ :- \ g_a \rightarrow c_a \\ \vdots \end{array} \right)$$

Figure 4.1 Notation for action system $\mathcal{A}$

and must establish the invariant. An action $a \in A$, is said to be enabled when its guard g_a evaluates to *true*. When an action system is executed the initialisation is performed first. Execution proceeds by repeated selection of an enabled guard and execution of the associated command. If no action is enabled, the action system is deadlocked. An action system is said to abort if its initialisation or one of its actions aborts.

For a set of actions $X \subseteq A$ we write $gd(\mathcal{A}_X)$ for the disjunction of the guards of all actions in $\mathcal{A}_X$. Also, for any sequence s of elements of A we write $\mathcal{A}_s$ for the sequential composition of the $\mathcal{A}$ elements corresponding to s, with $\mathcal{A}_{\langle\rangle}$ defined as *skip*. It will often be clear from the context which action system $\mathcal{A}$ is under consideration and in this case we will write $gd(X)$ and s respectively. The following example shows the use of a basic action system.

Example 16 With action systems, sequencing of events is achieved by enabling guards when appropriate. The following action system initialises its state variable x to 0 and then allows the actions h and l to flip the value of x alternately:

$$\mathcal{A}1 \triangleq \left(\begin{array}{l} \textbf{var } x : \{0,1\} \\ \textbf{initially } x := 0 \\ \textbf{action } h \ :- \ x = 0 \rightarrow x := 1 \\ \textbf{action } l \ :- \ x = 1 \rightarrow x := 0 \end{array} \right)$$

4.1.3 Failures-divergences for action systems

The standard model for CSP is the failures-divergences model. For process P, the failures $fails(P)$ is the set of all pairs (tr, R) where tr is a possible trace of events for P and R is a set of events each of which P may refuse after trace tr has occurred. The divergences, $divs(P)$, are the traces after which P may behave chaotically. CSP definitions are given in Appendix A.

A correspondence between action systems and CSP was given by Morgan [93] who defined the failures and divergences of an action system:

Definition 6 *(Morgan) For an action system* $\mathcal{A} = (A, S, \mathcal{A}_{init}, \mathcal{A}_A)$*:*

traces$(\mathcal{A})$ *are those* $t \in A^*$ *satisfying:*
$\overline{wp}(\mathcal{A}_{\langle init \rangle^\frown t}, true)$

fails$(\mathcal{A})$ *are those* $t \in A^*, X \in \mathbf{P}\ A$ *satisfying:*
$\overline{wp}(\mathcal{A}_{\langle init \rangle^\frown t}, \neg\ gd(\mathcal{A}_X))$

divs$(\mathcal{A})$ *are those* $t \in A^*$ *satisfying:*
$\overline{wp}(\mathcal{A}_{\langle init \rangle^\frown t}, false)$

Here, as defined above, $gd(\mathcal{A}_X)$ is the disjunction of the guards of all the actions in X.

4.1.4 Examples of basic action systems

Example 17 Consider the action system $\mathcal{A}1$ from Example 16. The sequence $\langle h \rangle$, for example, is a trace of $\mathcal{A}1$ since :

$\overline{wp}(\langle init \rangle^\frown \langle h \rangle, true)$
$\equiv \overline{wp}(x := 0, \overline{wp}(x = 0 \rightarrow x := 1, true))$
$\equiv \overline{wp}(x := 0, x = 0)$
$\equiv true$

Also, $(\langle h \rangle, \{h\})$ is a failure since:

$$\begin{aligned}
&\overline{wp}(\langle init \rangle^\frown \langle h \rangle, x = 1) \\
&\equiv \overline{wp}(x := 0, \overline{wp}(x = 0 \rightarrow x := 1, x = 1)) \\
&\equiv \overline{wp}(x := 0, x = 0) \\
&\equiv true
\end{aligned}$$

In fact, $\mathcal{A}1$ is equivalent to process $P2$ in Example 3 from Chapter 3 in that it has the same failures and divergences. The actions h and l must alternate, starting with h. Therefore, after an h action, repetition of h is refused, and similarly for l. The system cannot diverge. Since $\mathcal{A}1$ is equivalent to $P2$ in this sense it could therefore also be considered secure with respect to Definition 2 but insecure against Definition 1. ♮

Example 18 Many other action systems are also equivalent to $P2$. Consider the following example in which a specification statement (see Appendix B) is used to specify the commands:

$$\mathcal{A}2 \mathrel{\widehat{=}} \left(\begin{array}{l}
\textbf{var } s : \text{seq } \mathbb{N};\ n : \mathbb{N} \\
\textbf{initially } [s = \langle\rangle \wedge n \in \mathbb{N}] \\
\textbf{action } h \ :- \ s, n : [s_0 = \langle\rangle \wedge n > n_0 \wedge s = \langle n \rangle] \\
\textbf{action } l \ :- \ s : [s_0 \neq \langle\rangle \wedge s = \langle\rangle]
\end{array} \right)$$

$\mathcal{A}2$ looks rather different to $\mathcal{A}1$, yet the action l depends on h in exactly the same way. Notice that $\mathcal{A}2$ allows nondeterminism in the choice of n made initially and on execution of h. However, from the point of view of its CSP semantics, $\mathcal{A}2$ is deterministic. This is because the nondeterministic assignments cannot at any stage alter the set of guards which are enabled. ♮

These two examples show that there can be a great difference between two systems which are nevertheless regarded as equivalent with respect to CSP failures- divergence. The action systems specifications make these differences clear and present the possibility of data refinement.

Example 19 In contrast to the two action systems above, $\mathcal{A}3$ allows internal choice which does alter the available actions. This is therefore nondeterministic in the CSP sense.

$$\mathcal{A}3 \mathrel{\widehat{=}} \left(\begin{array}{l} \textbf{var } x : \{0,1\} \\ \textbf{initially } x := 0 \\ \textbf{action } h \ :- \ x = 0 \rightarrow x :\in \{0,1\} \\ \textbf{action } l \ :- \ x = 1 \rightarrow x := 0 \end{array} \right)$$

The sequence $\langle h, h \rangle$ is a possible trace of $\mathcal{A}3$ (in the case that the first h choses to set x to 0). However, $(h, \{h\})$ is also a failure since h may arbitrarily choose to set x to 1. Hence the failures-divergences interpretation is nondeterministic. $\mathcal{A}3$ is equivalent to the CSP process:

$$Q3 = h \rightarrow Q3a$$

$$\text{where } \ Q3a = (h \rightarrow Q3a \sqcap l \rightarrow Q3)$$

which fails to satisfy both Definitions 1 and 2. ♮

4.1.5 A choice operator for actions

The following choice operator for actions is defined by Sinclair and Woodcock [123].

Definition 7 *For actions* $a1$ *and* $a2$*:*

$$wp(a1 \parallel a2, \alpha) \equiv wp(a1, \alpha) \wedge wp(a2, \alpha)$$

It follows that:

$$\overline{wp}(a1 \parallel a2, \alpha) \equiv \overline{wp}(a1, \alpha) \vee \overline{wp}(a2, \alpha)$$

This represents an internal choice between actions, so both paths must establish the desired postcondition. The combined action is enabled when either of the component actions is enabled:

$$\overline{wp}(a1 \parallel a2, true)$$
$$\equiv \overline{wp}(a1, true) \vee \overline{wp}(a2, true)$$
$$\equiv gd(a1) \vee gd(a2)$$

In fact, as confirmed by their weakest preconditions, the choice between naked guarded commands is equivalent to an "alternative" command in Dijkstra's guarded command language with the appropriate guard:

$$(g1 \to c1 \,[\!]\, g2 \to c2) \equiv (g1 \vee g2) \to \begin{array}{ll} \textbf{if} & g1 \to c1 \\ [\!] & g2 \to c2 \\ \textbf{fi} & \end{array}$$

The operator is used in the definitions of hiding and of lazy deterministic security for action systems. Amongst the properties enjoyed by the choice operator are the following distribution properties:

Property 2 *For guarded commands* $a1, a2, a3$*:*

$$a1; (a2 \,[\!]\, a3) \equiv (a1; a2) \,[\!]\, (a1; a3)$$
$$(a1 \,[\!]\, a2); a3 \equiv (a1; a3) \,[\!]\, (a2; a3)$$

4.1.6 Infinite traces

The failures-divergences model of CSP cannot deal properly with unbounded nondeterminism since it cannot distinguish between a process which may never terminate and one which is guaranteed to terminate but after an unknown and unbounded number of events. For example, processes Q and R below have the same failures and divergences.

$$Q \mathrel{\widehat{=}} STOP \sqcap l \to Q$$

$$R \mathrel{\widehat{=}} \sqcap i \in \mathbf{N} \bullet R_i$$
where $R_{i+1} \mathrel{\widehat{=}} l \to R_i$ and $R_0 = STOP$

The solution in action systems as in CSP is to extend the model to include the consideration of infinite traces. Butler [20] defined infinite traces for action systems and demonstrated the correspondence with CSP.

There are some circumstances in which it is necessary to move to the infinite traces model to represent the action systems/CSP correspondence correctly. The problem arises in the definition of the hiding operator and is caused by the presence of unbounded nondeterminism as displayed by R. For

example, hiding l in either of Q or R leads to divergence in the CSP failures-divergences semantics. However, hiding l in an action system equivalent to R (by consideration of failures- divergence) would lead to deadlock rather than divergence.

For a correct treatment of the hiding operator it is therefore necessary to use the infinite traces model. If σ is an infinite sequence of actions then we write $\overline{inf}(\sigma)$ to characterise the set of states from which execution of the whole of σ is possible. Infinite sequential composition of actions was formalised by Butler and Morgan [22] allowing the following definition:

Definition 8 *(Butler and Morgan) For action system* $\mathcal{A} = (A, S, \mathcal{A}_{init}, \mathcal{A}_A)$ *the infinite traces are those* $t \in A^\omega$ *satisfying:*

$$\overline{inf}(\mathcal{A}_{(init)}{}^\frown t)$$

4.2 Action systems with internal actions

The CSP eager deterministic security property was defined using the CSP hiding operator. To make a direct translation of this property into action systems an equivalent action systems operator will be needed. This section follows Butler [20] in describing action systems with internal actions. This paves the way for a definition of action system hiding.

In an action system, an action which is hidden becomes an internal action. An internal action may be executed any number of times between occurrences of visible actions. It may change the state, and it may move the action system into a state in which a different set of actions is enabled. If it is possible for internal actions to occur infinitely many times in succession then the action system is said to diverge.

4.2.1 An iterative command

To reflect the way in which an internal action may be performed repeatedly between visible actions, the iterative command, **it** a **ti**, defined by Butler and Morgan [22] is used. This allows repetition of a with the possibility

of terminating with *skip* at each iteration. The definition makes use of the least fixed point operator μ (see Appendix B), here used with respect to the refinement ordering (described in Chapter 6).

Definition 9 *(Butler and Morgan)*

$$\textbf{it } a \textbf{ ti} \mathrel{\widehat{=}} (\mu X \bullet \mathit{skip} \,\|\, (a;\ X))$$

Unwinding the definition for iteration gives:

$$\begin{aligned}
&\textbf{it } a \textbf{ ti} \\
&= \mathit{skip} \,\|\, (a;\ (\textbf{it } a \textbf{ ti})) && \text{[Definition 9]} \\
&= \mathit{skip} \,\|\, (a;\ (\mathit{skip} \,\|\, (a;\ (\textbf{it } a \textbf{ ti})))) && \text{[Definition 9]} \\
&= \mathit{skip} \,\|\, (a;\ \mathit{skip}) \,\|\, (a;\ a;\ (\textbf{it } a \textbf{ ti})) && \text{[Property 2]} \\
&= \mathit{skip} \,\|\, a \,\|\, (a;\ a;\ (\textbf{it } a \textbf{ ti})) && \text{[Property of } \mathit{skip}\text{]} \\
&= \mathit{skip} \,\|\, a \,\|\, (a;\ a) \,\|\, (a;\ a;\ a;\ (\textbf{it } a \textbf{ ti})) && \text{[Definition 9]}
\end{aligned}$$

If for a particular a it can be shown that iterating a k times is miraculous, then the above expression can be simplified by removing all sequences of length $\geq k$. This was shown in detail by Sinclair and Woodcock [123]. Following this unwinding it can be seen informally that the weakest precondition for **it** a **ti** to establish postcondition α is:

$$wp(\mathit{skip}, \alpha) \wedge wp(a, \alpha) \wedge wp(a;\ a, \alpha) \wedge \cdots$$

This motivates the formal definition which uses the least fixed point operator on predicates with respect to logical implication:

Definition 10 *(Butler and Morgan)*

$$wp(\textbf{it } a \textbf{ ti}, \alpha) = (\mu Y \bullet \alpha \wedge wp(a, Y))$$

The iteration of a set X of actions from action system $\mathcal{A}$ is:

$$IT_X = \textbf{it } (\mathop{\Big\|}_{x \in X} \mathcal{A}_x) \textbf{ ti}$$

4.2.2 Internal actions and CSP correspondence

Action systems may now include a set I of internal actions. This is denoted:

$$\mathcal{A} = (A, S, \mathcal{A}_{init}, \mathcal{A}_A, \mathcal{A}_I)$$

The definition of correspondence with CSP must be extended to include the possible occurrence of internal actions. This definition and the following ones by Butler may be found in reference [20].

Definition 11 *(Butler) For action system $\mathcal{A} = (A, S, \mathcal{A}_i, \mathcal{A}_A, \mathcal{A}_I)$ with hidden actions, the failures are those $t \in A^*$, $X \subseteq A$ satisfying:*

$$\overline{wp}(\mathcal{A}_{\langle i \rangle^\frown t} * I, \neg\ gd_I(\mathcal{A}_X))$$

where $\mathcal{A}_s * I$ is defined by:

$$\mathcal{A}_{\langle\rangle} * I = skip$$
$$\mathcal{A}_{\langle a \rangle} * I = \mathcal{A}_a;\ IT_I$$
$$\mathcal{A}_{s^\frown t} * I = (\mathcal{A}_s * I);\ (\mathcal{A}_t * I)$$

and

$$gd_I(\mathcal{A}_X) = (\bigvee x \in X \bullet gd(IT_I;\ \mathcal{A}_x))$$

The divergences are those $t \in A^*$ satisfying:

$$\overline{wp}(\mathcal{A}_{\langle i \rangle^\frown t} * I, false)$$

The infinite traces are those are those $\sigma \in A^\omega$ satisfying:

$$\overline{inf}(\mathcal{A}_{\langle i \rangle^\frown \sigma} * I)$$

The execution of the visible actions of an action system may now be interspersed with any sequence of hidden actions which is enabled.

4.2.3 Hiding for action systems

Hiding a set X of actions in action system $\mathcal{A}$ is simply a matter of removing X actions from the set of visible actions of $\mathcal{A}$ and including them in the internal actions. This prompts the following definition:

Definition 12 *(Butler) If* $\mathcal{A} = (A, S, \mathcal{A}_{init}, \mathcal{A}_A, \mathcal{A}_I)$ *and* $X \subseteq A$ *then:*

$$\mathcal{A} \setminus X \mathrel{\widehat=} (A - X, S, \mathcal{A}_{init}, \mathcal{A}_{A-X}, \mathcal{A}_{I \cup X})$$

The following theorem shows correspondence with CSP in the infinite traces model for the hiding operator.

Theorem 1 *(Butler) If* $\mathcal{A} = (A, S, \mathcal{A}_{init}, \mathcal{A}_A, \mathcal{A}_I)$ *and* $X \subseteq A$ *then:*

$$FDI(\mathcal{A} \setminus X) = FDI(\mathcal{A}) \setminus X$$

Here, $FDI(\mathcal{A})$ denotes the failures, divergence and infinite traces representation of action system $\mathcal{A}$. The notation $FD(\mathcal{A})$ will also be used and this denotes the failures-divergences representation of $\mathcal{A}$.

Example 20 As an example of the use of hiding in action systems consider the following specification of a component intended to monitor a user's password to check that it is not older than the maximum allowed age for the system (in this case, 100 days). If a positive number of days is left before expiry, the action *newday* decrements the variable *daysleft* (this could be thought of as acting in parallel with a clock which determines when a new day occurs). When the password has expired the system waits for the user to cooperate in resetting the time period (presumably in conjunction with entering a new password). Until this is done it will refuse to participate in the *newday* action.

$$Countdown \mathrel{\widehat=}$$
$$\left(\begin{array}{l} \textbf{var } daysleft : \mathbb{N} \\ \textbf{initially } daysleft := 100 \\ \textbf{action } newday \ :- \ daysleft > 0 \rightarrow daysleft := daysleft - 1 \\ \textbf{action } reset \ :- \ daysleft = 0 \rightarrow daysleft := 100 \end{array}\right)$$

The two actions in this system are both external. The system waits for the cooperation of the user to reset the time. However, suppose the component is intended for a system which generates passwords internally. The time counter can then be reset internally. This can be represented by hiding the *reset* action.

$$Countdown \setminus \{reset\} \mathrel{\widehat{=}} \left(\begin{array}{l} \textbf{var } daysleft : \mathbb{N} \\ \textbf{initially } daysleft := 100 \\ \textbf{action } newday \ :- \ daysleft > 0 \rightarrow daysleft := daysleft - 1 \\ \textbf{internal } reset \ :- \ daysleft = 0 \rightarrow daysleft := 100 \end{array} \right)$$

This system will never refuse the *newday* action since resetting is done internally when the *reset* action is enabled. The internal action cannot occur more than once because its guard becomes false after execution. So there is no possibility of divergence in this system. ♮

Example 21 The action system $\mathcal{A}1$ in Example 16 was shown to correspond to the process $P2$ from Example 3. Here the effects of hiding h are examined. The resulting system is given by:

$$\mathcal{A}1 \setminus \{h\} \mathrel{\widehat{=}} \left(\begin{array}{l} \textbf{var } x : \{0,1\} \\ \textbf{initially } x := 0 \\ \textbf{action } l \ :- \ x = 1 \rightarrow x := 0 \\ \textbf{internal } h \ :- \ x = 0 \rightarrow x := 1 \end{array} \right)$$

The internal action h may be iterated between each visible action of the system. However, one h followed by another would be miraculous since:

$$\begin{aligned} & wp(x = 0 \rightarrow x := 1, wp(x = 0 \rightarrow x := 1, false)) \\ & \equiv wp(x = 0 \rightarrow x := 1, x = 0 \Rightarrow false) \\ & \equiv wp(x = 0 \rightarrow x := 1, x \neq 0) \\ & \equiv x = 0 \Rightarrow 1 \neq 0 \\ & \equiv true \end{aligned}$$

So, using the simplification of it h ti for miraculous branches:

$$\textbf{it } h \textbf{ ti} \equiv \mathit{skip} \,[\!]\, h$$

It might appear that it h ti could choose to skip, in which case action l would never become enabled. However, by the way the iteration construct is defined, it h ti is forced to take a path which enables l. This can be seen by calculating the guard of l:

$$\begin{aligned}
& gd_h(l) \\
& \equiv gd(\textbf{it } h \textbf{ ti};\ l) && \text{[Defn. 11]} \\
& \equiv \overline{wp}(\textbf{it } h \textbf{ ti};\ l, \mathit{true}) \\
& \equiv \overline{wp}((\mathit{skip} \,[\!]\, h);\ l, \mathit{true}) \\
& \equiv \overline{wp}(\mathit{skip} \,[\!]\, h, x = 1) \\
& \equiv \overline{wp}(\mathit{skip}, x = 1) \vee \overline{wp}(h, x = 1) \\
& \equiv x = 1 \vee x = 0 \\
& \equiv \mathit{true}
\end{aligned}$$

$\natural$

4.3 Communication in action systems

Action systems communicate with their environment using input and output actions. This section gives definitions which take account of value-passing in action systems. Again, this correspondence was first defined by Butler [20]. A value-passing action system is denoted:

$$\mathcal{A} = (A, S, \mathcal{A}_{init}, \mathcal{A}_A, \mathcal{A}_I, \mathit{dir})$$

where dir is a total function from A to the set $\{\mathit{in}, \mathit{out}\}$. A can be thought of as the set of channel names, with each action labelled by a channel name and each channel declared to be either an input or an output channel.

If a is an input channel then $a.i$ represents the input of value i on channel a, and similarly for an output channel. If $\mathcal{W}$ is the set of all possible values

that might be communicated, then the alphabet of action system $\mathcal{A}$ is $\mathcal{A}_W$ where:

$$\mathcal{A}_W \triangleq \{a.i \mid a \in A \wedge i \in W\}$$

For any set of values $X \subseteq W$ we write $\mathcal{A}_{a.X}$ for the set of all input actions on channel a with input values from X.

Example 22 The following example shows the notation used for a value-passing action system. System $\mathcal{A}5$ defines a stack which can always accept input and which, when non-empty, will output the value most recently entered.

$$\mathcal{A}5 \triangleq \left(\begin{array}{l} \textbf{var } s : \text{seq } \mathbb{N} \\ \textbf{initially } s := \langle\rangle \\ \textbf{action } \textit{put\_on} \textbf{ in } i? : \mathbb{N} \ :- \\ \quad \textit{true} \rightarrow s := s^\frown\langle i?\rangle \\ \textbf{action } \textit{take\_off} \textbf{ out } o! : \mathbb{N} \ :- \\ \quad s \neq \langle\rangle \rightarrow s, o! := \textit{front } s, \textit{last } s \end{array} \right)$$

When s is non-empty, both guards are enabled and so both the input action *put_on* and and the output action *take_off* are possible. This is viewed as an external choice and does not introduce nondeterminism to the system. $\mathcal{A}5$ is equivalent to the CSP process:

$$\begin{aligned} STK &= STK_{\langle\rangle} \\ STK_{\langle\rangle} &= \textit{put\_on}?i \rightarrow STK_{\langle i\rangle} \\ STK_{s^\frown\langle i\rangle} &= (\textit{put\_on}?j \rightarrow STK_{s^\frown\langle i,j\rangle}) \ \Box \ (\textit{take\_off}!i \rightarrow STK_s) \end{aligned}$$

$\natural$

4.3.1 Value-passing actions

Given a particular value, $v \in W$, the action corresponding to the input of v on channel a is calculated from $\mathcal{A}_a$ using the following definition in which $i?$ is the variable representing the input value:

Definition 13 *(Butler) If* $dir(a) = in$ *then:*

$$\mathcal{A}_{a.v} \mathrel{\widehat{=}} \mathcal{A}_a[\textbf{value } v/i?]$$

Here, substitution by value is used where, for x not free in α:

$$wp(S[\textbf{value } E/x], \alpha) \mathrel{\widehat{=}} wp(S, \alpha)[E/x]$$

For example:

$$\begin{aligned}
&\mathcal{A}5_{put_on.3} \\
&\equiv (true \rightarrow s := s^\frown\langle i?\rangle)[\textbf{value } 3/i?] \\
&\equiv true \rightarrow s := s^\frown\langle 3\rangle
\end{aligned}$$

A similar means is used to calculate an output action for a given output value v with $o!$ being the variable representing the output of the channel:

Definition 14 *(Butler) If* $dir(a) = out$ *then:*

$$\mathcal{A}_{a.v} \mathrel{\widehat{=}} (\textbf{local } o! \bullet \mathcal{A}_a[o! = v])$$

Here, $o!$ is introduced as a local variable where for $o!$ not free in α:

$$wp((\textbf{local } o! \bullet s), \alpha) \mathrel{\widehat{=}} (\forall\, o! \bullet wp(s, \alpha))$$

Again, taking the stack specification:

$$\begin{aligned}
&\mathcal{A}5_{take_off.2} \\
&\equiv (\textbf{local } o! \bullet s \neq \langle\rangle \rightarrow s, o! := front\ s, last\ s)[o! = 2] \\
&\equiv s \neq \langle\rangle \rightarrow (\textbf{local } o! \bullet o! := last\ s[o! = 2]);\ s := front\ s \\
&\equiv s \neq \langle\rangle \rightarrow (\textbf{local } o! \bullet last\ s = 2 \rightarrow o! := last\ s);\ s := front\ s \\
&\equiv s \neq \langle\rangle \wedge last\ s = 2 \rightarrow s := front\ s
\end{aligned}$$

4.3.2 Value-passing action systems and CSP

For an input channel a and $X \subseteq \mathcal{W}$, the set $\mathcal{A}_{a.X}$ is enabled whenever some value from X satisfies the guard of $\mathcal{A}_a$. However, an output action may be

required to select a nondeterministic output value. In this case, a particular value may be possible (and hence the action enabled) but is not selected by the action. Thus that value is refused. This motivates the following definitions which can be used to describe the failures of a value-passing action system:

Definition 15 *(Butler) If* $dir(a) = in$*, then for* $X \subseteq \mathcal{W}$*:*

$$commgd(\mathcal{A}_{a.X}) \mathrel{\widehat{=}} \exists x \in X \bullet gd(\mathcal{A}_a)$$

If $dir(a) = out$ *with output variable* y*, then for* $X \subseteq \mathcal{W}$*:*

$$commgd(\mathcal{A}_{a.X}) \mathrel{\widehat{=}} gd(\mathcal{A}_a) \wedge wp(\mathcal{A}_a, y \in X)$$

If $\mathcal{X} \subseteq \mathcal{A}_{\mathcal{W}}$*, possibly containing inputs and outputs on several channels then:*

$$commgd(\mathcal{A}_{\mathcal{X}}) \mathrel{\widehat{=}} (\bigvee a \in A \bullet commgd(\mathcal{A}_{\mathcal{X}\restriction a}))$$

For an action system with internal actions, I:

$$commgd_I(\mathcal{A}_{\mathcal{X}}) \mathrel{\widehat{=}} (\bigvee a \in A \bullet \overline{wp}(IT_I, commgd(\mathcal{A}_{\mathcal{X}\restriction a})))$$

where $\mathcal{X} \restriction a$ *is the set of all* $\mathcal{X}$ *actions on channel* a*.*

Example 23 The output action a defined below makes an internal choice from the set $s : \mathbf{P}\ \mathbf{N}$ when selecting its output.

$$\textbf{action } a \textbf{ out } o! : \mathbf{N} \ :- \ s \neq \emptyset \rightarrow o! :\in s$$

Suppose $s = \{0, 1\}$, then a is guaranteed to output one of these values but could refuse either 0 or 1. This is reflected in the *commgd* for these since for $X \subseteq \mathbf{N}$:

$$\begin{aligned} &commgd(a.X) \equiv s \neq \emptyset \wedge wp(a, o! \in X) \\ &\equiv s \neq \emptyset \wedge s \subseteq X \end{aligned}$$

In the situation described, this is *true* when $X = \{0, 1\}$ (or any subset of $\mathbf{N}$ containing 0 and 1) but *false* for $X = \{0\}$ and $X = \{1\}$ (and any subset of $\mathbf{N}$ which fails to contain both 0 and 1). The action a cannot be guaranteed to output any one particular value unless s is a singleton set. ♮

It is now possible to define the correspondence with CSP up to infinite traces for value-passing action systems.

Definition 16 *(Butler) For action system* $\mathcal{A} = (A, S, \mathcal{A}_{init}, \mathcal{A}_A, \mathcal{A}_I, dir)$*:*

fails$(\mathcal{A})$ *are those* $t \in A^*_{\mathcal{W}}, X \subseteq A_{\mathcal{W}}$ *satisfying:*

$$\overline{wp}(\mathcal{A}_{\langle init \rangle^\frown t} * I, \neg\ commgd_I(\mathcal{A}_X))$$

divs$(\mathcal{A})$ *are those* $t \in A^*_{\mathcal{W}}$ *satisfying:*

$$\overline{wp}(\mathcal{A}_{\langle init \rangle^\frown t} * I, false)$$

infs$(\mathcal{A})$ *are those* $u \in A^\omega_{\mathcal{W}}$ *satisfying:*

$$\overline{inf}(\mathcal{A}_{\langle init \rangle^\frown u} * I)$$

It is a requirement for a well-formed action system that output actions should always terminate.

4.4 Parallel composition of action systems

A parallel composition operator for actions sytems was given by Butler [20]. This was shown to be equivalent to parallel composition in CSP. Common actions of the two systems are joined as described below and their states are unioned. The restriction is made that systems being joined by parallel composition should have no state variables in common. (It would be possible to define a more general composition but this would be at the expense of other pleasing features: in particular, the ability to refine and further decompose individual component systems independently would be lost).

First, the definition for systems without value-passing is given.

Definition 17 *(Butler) For action systems* $\mathcal{A} \mathrel{\widehat{=}} (A, S, \mathcal{A}_i, \mathcal{A}_A, \mathcal{A}_I)$ *and* $\mathcal{B} \mathrel{\widehat{=}} (B, T, \mathcal{B}_i, \mathcal{B}_B, \mathcal{B}_J)$ *the parallel composition is defined:*

$$\mathcal{A} \parallel \mathcal{B} \mathrel{\widehat{=}} (A \cup B, (S, T), \mathcal{A}_i \parallel \mathcal{B}_i, par(\mathcal{A}_A, \mathcal{B}_B), \mathcal{A}_I \cup \mathcal{B}_J)$$

where:

$$par(\mathcal{A}_A, \mathcal{B}_B) \mathrel{\widehat{=}} \mathcal{A}_{A-B} \cup \mathcal{B}_{B-A} \cup \{\mathcal{A}_c \parallel \mathcal{B}_c \mid c \in A \cap B\}$$

and:

$$(x1 := E1 \parallel x2 := E2) \mathrel{\widehat{=}} (x1, x2 := E1, E2)$$
$$(g1 \rightarrow c1) \parallel (g2 \rightarrow c2) \mathrel{\widehat{=}} (g1 \wedge g2 \rightarrow c1;\ c2)$$

When value-passing action systems are placed in parallel similarly-named channels must be combined. Following the approach of [20], two input channels may be combined in parallel as long as the types of the input variable are compatible. The resulting action will itself be an input action willing to accept any input suitable for both component actions.

Definition 18 *(Butler) For input actions:*

$$(\textbf{action } \mathcal{A}_a \textbf{ in } x? : X :- g \rightarrow c) \parallel (\textbf{action } \mathcal{B}_a \textbf{ in } x? : X' :- g' \rightarrow c')$$
$$\mathrel{\widehat{=}} (\textbf{action } (\mathcal{A} \parallel \mathcal{B})_a \textbf{ in } x? : (X \cap X') :- g \wedge g' \rightarrow c \parallel c')$$

When an output channel is combined with an input channel the output value of the former is taken as input for the latter. Again, any value that might be output by the output channel must be a possible input for the input channel, so the type of the output value must be a subset of the type of the input value. As well as being passed to the input action, the output value is still made externally visible. This means that the combined action is classed as an output action.

Definition 19 *(Butler) For output action and input action with $X \subseteq X'$:*

$$(\textbf{action } \mathcal{A}_a \textbf{ out } y! : X :- g \rightarrow c) \parallel (\textbf{action } \mathcal{B}_a \textbf{ in } x? : X' :- g' \rightarrow c')$$
$$\mathrel{\widehat{=}} (\textbf{action } (\mathcal{A} \parallel \mathcal{B})_a \textbf{ out } y! : X :- g \wedge g' \rightarrow c \parallel c' [\textbf{value } y!/x?])$$

where

$$wp(\mathcal{A}_a, y! \in X) \equiv true$$

and

$$(\exists i : X \bullet gd(\mathcal{B}_a[\textbf{value } i/x?]))$$
$$\Rightarrow (\forall i : X \bullet gd(\mathcal{B}_a[\textbf{value } i/x?]))$$

The first condition ensures that $\mathcal{A}_a$ outputs a value within the specified type (it also implies that $\mathcal{A}_a$ terminates). The second condition demands that if $\mathcal{B}_a$ is enabled for some input value then it is enabled for all possible input values.

Combining output channels could cause deadlock if the channels attempt to output differing values. Rather than attempting to define suitable conditions, the parallel combination of output actions will not be used.

Given the previous two definitions, parallel composition can be defined for value- passing action systems:

Definition 20 *(Butler) For action systems* $\mathcal{A} \triangleq (A, S, \mathcal{A}_i, \mathcal{A}_A, \mathcal{A}_I, dir_A)$ *and* $\mathcal{B} \triangleq (B, T, \mathcal{B}_i, \mathcal{A}_B, \mathcal{B}_J, dir_B)$ *which have no common state variables, no common output channels and which satisfy the conditions of Definition 19 for each input-output pair:*

$$\mathcal{A} \,||\, \mathcal{B} \triangleq (A \cup B, (S, T), \mathcal{A}_i \,||\, \mathcal{B}_i, \mathcal{A}_A \,||\, \mathcal{B}_B, \mathcal{A}_I \cup \mathcal{A}_J, dir)$$

in which commonly labelled actions are combined according to Definitions 18 and 19 and actions with labels unique to either A or B are included unchanged.

4.5 Nondeterminism in action systems

The previous sections in this chapter set out the basic definitions needed to formulate security conditions directly for action systems. The concept underlying the CSP security properties of Definitions 1 and 2 is that of nondeterminism. This section considers nondeterminism in action systems, giving a definition of determinism which is shown to correspond to determinism in CSP. The definition for the simple case of basic action systems is given first, with internal actions and value-passing taken account of later.

4.5.1 Determinism for simple action systems

In CSP a process P is deterministic if it does not diverge and cannot choose to refuse an event in which it might engage, that is:

$$divs(P) = \emptyset \wedge$$
$$tr^\frown\langle x\rangle \in traces(P) \Rightarrow (tr, \{x\}) \notin fails(P)$$

For an action system, nondeterminism in the failures-divergences sense arises if, after a particular sequence of events, it is possible both that some action will be enabled or that it will not have been. For example, consider action system $\mathcal{A}3$ from Example 19:

$$\mathcal{A}3 \mathrel{\widehat{=}} \left(\begin{array}{l} \textbf{var } x : \{0,1\} \\ \textbf{initially } x := 0 \\ \textbf{action } h \ :- \ x = 0 \rightarrow x :\in \{0,1\} \\ \textbf{action } l \ :- \ x = 1 \rightarrow x := 0 \end{array}\right)$$

Here, action h makes a nondeterministic choice for the value of x. If 1 is selected then action l is enabled, so $\langle h, l\rangle$ is a possible trace. However, if 0 is selected then l will be refused, so $(h, \{l\})$ is a failure. An action system is therefore nondeterministic if it is possible for a trace both to establish a guard and to establish the negation of the guard. This motivates the following definition.

Definition 21 *Action system $\mathcal{A} = (A, S, \mathcal{A}_{init}, \mathcal{A}_A)$ is deterministic if $divs(\mathcal{A}) = \emptyset$ and for each trace tr and action x:*

$$\overline{wp}(\mathcal{A}_{\langle init\rangle^\frown tr}, gd(\mathcal{A}_x)) \equiv wp(\mathcal{A}_{\langle init\rangle^\frown tr}, gd(\mathcal{A}_x))$$

This definition says that if a trace may enable a guard, then it must enable that guard. For example, with $\mathcal{A}3$:

$$\overline{wp}(\mathcal{A}3_{\langle init,h\rangle}, x = 1)$$
$$\equiv \overline{wp}(x := 0, x = 0 \wedge \overline{wp}(x :\in \{0,1\}, x = 1))$$
$$\equiv \overline{wp}(x := 0, true)$$
$$\equiv true$$

whereas:

$$wp(\mathcal{A}3_{\langle init,h\rangle}, x = 1)$$
$$\equiv wp(x := 0, x = 0 \Rightarrow wp(x :\in \{0,1\}, x = 1))$$
$$\equiv \overline{wp}(x := 0, x = 0 \Rightarrow false)$$
$$\equiv false$$

Definition 21 gives the criterion for determinism in the failures- divergence model but this would be exactly the same for a system considered with respect to infinite traces too.

The following theorem shows how Definition 21 corresponds to CSP determinism.

Theorem 2 *If $FD(\mathcal{A})$ stands for the failures-divergences semantics of action system $\mathcal{A} = (A, S, \mathcal{A}_{init}, \mathcal{A}_A)$ then :*

$$\mathcal{A} \textit{ deterministic } \Leftrightarrow FD(\mathcal{A}) \textit{ deterministic}$$

Proof Assuming $FD(\mathcal{A})$ is deterministic it follows immediately from Definition 6 that for each trace tr and action x:

$$\overline{wp}(\mathcal{A}_{\langle i\rangle^\frown tr^\frown\langle x\rangle}, true) \Rightarrow \neg\ \overline{wp}(\mathcal{A}_{\langle i\rangle^\frown tr}, \neg\ gd(\mathcal{A}_x))$$

Using wp definitions this is equivalent to:

$$\overline{wp}(\mathcal{A}_{\langle i\rangle^\frown tr}, gd(\mathcal{A}_x)) \Rightarrow wp(\mathcal{A}_{\langle i\rangle^\frown tr}, gd(\mathcal{A}_x))$$

Also:

$$wp(\mathcal{A}_{\langle i\rangle^\frown tr}, gd(\mathcal{A}_x)) \Rightarrow \overline{wp}(\mathcal{A}_{\langle i\rangle^\frown tr}, gd(\mathcal{A}_x))$$
$$\equiv \overline{wp}(\mathcal{A}_{\langle i\rangle^\frown tr}, \neg\ gd(\mathcal{A}_x)) \vee \overline{wp}(\mathcal{A}_{\langle i\rangle^\frown tr}, gd(\mathcal{A}_x))$$
$$\equiv \overline{wp}(\mathcal{A}_{\langle i\rangle^\frown tr}, true) \qquad [\vee\text{- dist. Prop. 1}]$$

which holds since tr is a trace. So:

$$FD(\mathcal{A}) \text{ deterministic} \Rightarrow \mathcal{A} \text{ deterministic}$$

Similarly,

$$\mathcal{A} \text{ deterministic} \Rightarrow FD(\mathcal{A}) \text{ deterministic}$$

follows from the definitions and *wp* calculus. ♮

Definition 21 is a condition which requires proof for all possible traces of the system. It might be hoped that a set of sufficient conditions similar to unwinding could be found which would reduce the task to checking the effect of each action on each guard. However, this is complicated by the fact that a nondeterministic choice might not affect the set of guards enabled until a later stage. For example:

$$\mathcal{A}6 \mathrel{\widehat{=}} \left(\begin{array}{l} \textbf{var } x : \mathbb{N} \\ \textbf{initially } x := 0 \\ \textbf{action } set \ :- \ x = 0 \rightarrow x : [x \geq 100] \\ \textbf{action } dec \ :- \ x \geq 1 \rightarrow x : [x = x_0 - 1] \end{array} \right)$$

The action *set* will always enable the guard of *dec*. The nondeterminism becomes apparent only after at least 100 repetitions of *dec*.

As shown by Theorem 3, if the initialisation and each action is deterministic then the overall system is deterministic. These conditions are very strong, but are certainly sufficient for the determinism of an action system.

Theorem 3 *For action system $\mathcal{A} = (A, S, \mathcal{A}_i, \mathcal{A}_A)$, if for any postcondition α:*

$$\overline{wp}(\mathcal{A}_{init}, \alpha) \equiv wp(\mathcal{A}_{init}, \alpha)$$

and for each $a \in A$ and postcondition α:

$$\overline{wp}(\mathcal{A}_{tr}, \alpha) \equiv wp(\mathcal{A}_{tr}, \alpha)$$

then $\mathcal{A}$ is deterministic.

Proof By induction on length of trace we show that for any trace *tr* and postcondition α

$$\overline{wp}(\mathcal{A}_a, \alpha) \equiv wp(\mathcal{A}_a, \alpha)$$

Base case:

$$\overline{wp}(\mathcal{A}_{init}, \alpha) \equiv wp(\mathcal{A}_{init}, \alpha)$$

follows directly from assumption on $\mathcal{A}_{init}$.
Inductive step: assume that for all α:

$$\overline{wp}(\mathcal{A}_{\langle init\rangle^\frown t}, \alpha) \equiv wp(\mathcal{A}_{\langle init\rangle^\frown t}, \alpha)$$

holds for all t with $\#t \leq n$. Then:

$$\begin{aligned}
&\overline{wp}(\mathcal{A}_{\langle init\rangle^\frown t^\frown \langle x\rangle}, \alpha) \\
&\equiv \overline{wp}(\mathcal{A}_{\langle init\rangle^\frown t}, \overline{wp}(\mathcal{A}_x, \alpha)) \\
&\equiv wp(\mathcal{A}_{\langle init\rangle^\frown t}, \overline{wp}(\mathcal{A}_x, \alpha)) && \text{[Ind. hyp.]} \\
&\equiv wp(\mathcal{A}_{\langle init\rangle^\frown t}, wp(\mathcal{A}_x, \alpha)) && \text{[Ass. on } \mathcal{A}_x\text{]} \\
&\equiv wp(\mathcal{A}_{\langle init\rangle^\frown t^\frown \langle x\rangle}, \alpha)
\end{aligned}$$

$\natural$

Whilst requiring each action to be deterministic is a sufficient condition for determinism of an action system it is certainly not necessary. This was shown, for example, by $\mathcal{A}2$ of Example 18.

Another approach to the proof of determinism is to note that, in a nondivergent system, all actions whose guard is *true* are always enabled. Thus it is sufficient to prove that the system is deterministic for those actions whose guards are some condition other than *true*. For value-passing channels, the guard of an action would be calculated for possible outputs. This is expressed in the following theorem.

Theorem 4 *For nondivergent action system $\mathcal{A} = (A, S, \mathcal{A}_i, \mathcal{A}_A)$ if there is some $A' \subseteq A$ such that $gd(a) \equiv true$ for all $a \in A'$ then $\mathcal{A}$ is deterministic if for each trace tr and each $x \in A - A'$:*

$$\overline{wp}(\mathcal{A}_{\langle i\rangle^\frown tr}, gd(x)) \equiv wp(\mathcal{A}_{\langle i\rangle^\frown tr}, gd(x))$$

Proof For any $a \in A'$ and trace tr:

$$\begin{aligned}
&wp(\mathcal{A}_{\langle i\rangle^\frown tr}, gd(a)) \\
&\equiv wp(\mathcal{A}_{\langle i\rangle^\frown tr}, true) \\
&\equiv \neg\ \overline{wp}(\mathcal{A}_{\langle i\rangle^\frown tr}, false) \equiv true && \text{[}\mathcal{A}\text{ nondivergent]}
\end{aligned}$$

also:

$$\begin{aligned}
&\overline{wp}(\mathcal{A}_{\langle i \rangle^\frown tr}, gd(a)) \\
&\equiv \overline{wp}(\mathcal{A}_{\langle i \rangle^\frown tr}, true) \\
&\equiv true && [tr \text{ is a trace}]
\end{aligned}$$

Hence:

$$wp(\mathcal{A}_{\langle i \rangle^\frown tr}, gd(a)) \equiv \overline{wp}(\mathcal{A}_{\langle i \rangle^\frown tr}, gd(a))$$

By assumption, this is also true for each action in $A - A'$. So for all $x \in A$:

$$wp(\mathcal{A}_{\langle i \rangle^\frown tr}, gd(x)) \equiv \overline{wp}(\mathcal{A}_{\langle i \rangle^\frown tr}, gd(x))$$

That is, $\mathcal{A}$ is deterministic. ♮

It follows from Theorem 4 that if all guards are *true* then $\mathcal{A}$ is deterministic.

Corollary 5 *If $\mathcal{A} = (A, S, \mathcal{A}_i, \mathcal{A}_A)$ is a nondivergent action system with $gd(a) = true$ for all $a \in A$ then $\mathcal{A}$ is deterministic.*

It is also worth noting that an action whose guard is never enabled by the system satisfies the condition for determinism:

Theorem 6 *For nondivergent action system $\mathcal{A} = (A, S, \mathcal{A}_i, \mathcal{A}_A)$, if $a \in A$ such that for each trace tr:*

$$\overline{wp}(\mathcal{A}_{\langle i \rangle^\frown tr}, gd(a)) \equiv false$$

then for each trace tr:

$$wp(\mathcal{A}_{\langle i \rangle^\frown tr}, gd(a)) \equiv \overline{wp}(\mathcal{A}_{\langle i \rangle^\frown tr}, gd(a))$$

Proof

$$\begin{aligned}
&true \equiv wp(\mathcal{A}_{\langle i \rangle^\frown tr}, true) && [tr \text{ a trace}] \\
&\equiv \overline{wp}(\mathcal{A}_{\langle i \rangle^\frown tr}, gd(a) \vee \neg\, gd(a)) \\
&\equiv \overline{wp}(\mathcal{A}_{\langle i \rangle^\frown tr}, gd(a)) \vee \overline{wp}(\mathcal{A}_{\langle i \rangle^\frown tr}, \neg\, gd(a)) && [\vee\text{-dist. Prop 1}] \\
&\equiv false \vee \overline{wp}(\mathcal{A}_{\langle i \rangle^\frown tr}, \neg\, gd(a)) && [\text{Ass.}] \\
&\equiv \overline{wp}(\mathcal{A}_{\langle i \rangle^\frown tr}, \neg\, gd(a)) \\
&\equiv \neg\, wp(\mathcal{A}_{\langle i \rangle^\frown tr}, gd(a))
\end{aligned}$$

Hence:

$$wp(\mathcal{A}_{\langle i \rangle \hat{} tr}, gd(a)) \equiv false \equiv \overline{wp}(\mathcal{A}_{\langle i \rangle \hat{} tr}, gd(a))$$

$\natural$

4.5.2 Determinism for action systems with internal actions

When internal actions are taken into account, a trace of the system consists of a sequence of visible actions, t, interspersed with internal actions, with the whole being a possible sequence of events within the action system. The system is nondeterministic if execution of some t interspersed with internal actions can enable a particular action, while for the same t interspersed with internal actions in a possibly different way, the guard of the action may also become false. This indicates that the definition of determinism for an action system should be extended.

Definition 22 *Action system* $\mathcal{A} = (A, S, \mathcal{A}_{init}, \mathcal{A}_A, \mathcal{A}_I)$ *is deterministic if* $divs(\mathcal{A}) = \emptyset$ *and for each possible trace of visible actions* $t \in A^*$ *and for each action* x*:*

$$\overline{wp}(\mathcal{A}_{\langle init \rangle \hat{} tr} * I, gd_I(\mathcal{A}_x)) = wp(\mathcal{A}_{\langle init \rangle \hat{} tr} * I, gd_I(\mathcal{A}_x))$$

Again, this definition can be shown to correspond to CSP.

Theorem 7 *For action system with internal actions* $\mathcal{A} = (A, S, \mathcal{A}_i, \mathcal{A}_A, \mathcal{A}_I)$*:*

$$\mathcal{A} \text{ deterministic} \Leftrightarrow FDI(\mathcal{A}) \text{ deterministic}$$

Proof Suppose $FDI(\mathcal{A})$ deterministic, then for each trace $t \in A^*$ and action x:

$$\overline{wp}(\mathcal{A}_{\langle i \rangle \hat{} t \hat{} \langle x \rangle} * I, true) \Rightarrow \neg\ \overline{wp}(\mathcal{A}_{\langle i \rangle \hat{} t} * I, \neg\ gd_I(\mathcal{A}_x))$$
$$\equiv \overline{wp}(\mathcal{A}_{\langle i \rangle \hat{} t} * I, gd_I(\mathcal{A}_x)) \Rightarrow wp(\mathcal{A}_{\langle i \rangle \hat{} t} * I, gd_I(\mathcal{A}_x))$$

This follows since:

$$
\begin{aligned}
&\overline{wp}(\mathcal{A}_{\langle i\rangle^\frown t^\frown\langle x\rangle} * I, true) \Rightarrow \\
&\equiv \overline{wp}((\mathcal{A}_{\langle i\rangle^\frown t} * I);\ (\mathcal{A}_{\langle x\rangle} * I), true) \\
&\equiv \overline{wp}(\mathcal{A}_{\langle i\rangle^\frown t} * I, \overline{wp}(\mathcal{A}_{\langle x\rangle} * I, true)) \\
&\equiv \overline{wp}(\mathcal{A}_{\langle i\rangle^\frown t} * I;\ I, \overline{wp}(\mathcal{A}_{\langle x\rangle} * I, true)) && [IT_X \equiv IT_X;\ IT_X] \\
&\equiv \overline{wp}(\mathcal{A}_{\langle i\rangle^\frown t} * I;\ I, \overline{wp}(\mathcal{A}_{\langle x\rangle}, \overline{wp}(I, true))) \\
&\equiv \overline{wp}(\mathcal{A}_{\langle i\rangle^\frown t} * I;\ I, \overline{wp}(\mathcal{A}_{\langle x\rangle}, true)) && [\overline{wp}(IT_X, true) \equiv true] \\
&\equiv \overline{wp}(\mathcal{A}_{\langle i\rangle^\frown t} * I, \overline{wp}(I;\ \mathcal{A}_{\langle x\rangle}, true)) \\
&\equiv \overline{wp}(\mathcal{A}_{\langle i\rangle^\frown t} * I, gd_I(\mathcal{A}_x))
\end{aligned}
$$

Also,

$$
\begin{aligned}
&wp(\mathcal{A}_{\langle i\rangle^\frown t} * I, gd_I(\mathcal{A}_x)) \Rightarrow \overline{wp}(\mathcal{A}_{\langle i\rangle^\frown t} * I, gd_I(\mathcal{A}_x)) \\
&\equiv \overline{wp}(\mathcal{A}_{\langle i\rangle^\frown t} * I, \neg\ gd_I(\mathcal{A}_x)) \vee \overline{wp}(\mathcal{A}_{\langle i\rangle^\frown t} * I, gd_I(\mathcal{A}_x)) \\
&\equiv \overline{wp}(\mathcal{A}_{\langle i\rangle^\frown t} * I, true)
\end{aligned}
$$

which is true since t is a trace and $FDI(\mathcal{A})$ deterministic ensures that the system does not diverge. Using this fact the proof follows as before. ♮

4.5.3 Determinism for value-passing action systems

For a value-passing action system it is important to consider not only whether an action is enabled for a particular channel but also what values it is possible to communicate. For example, an output on channel *take_off* of action system $\mathcal{A}5$ is enabled whenever the sequence s is non-empty. However, the output action $\mathcal{A}5_{take\_off.0}$ is possible exactly when the last value in s is 0.

Input actions, when enabled, are always ready to accept any value of the appropriate type, so there is no need to consider individual values. The question of type is important for both input and output channels since a channel can communicate a value of the appropriate type only. The *commgd* construct already takes account of these considerations and so this can be used in the following definition of determinism for value-passing action systems.

Definition 23 *Action system* $\mathcal{A} = (A, S, \mathcal{A}_{init}, \mathcal{A}_A, \mathcal{A}_I, dir)$ *is deterministic if* $divs(\mathcal{A}) = \emptyset$ *and for each trace* $t \in A^*$ *of visible actions and each* $x \in \mathcal{A}_W$*:*

$$\overline{wp}(\mathcal{A}_{\langle init \rangle^\frown tr} * I, gd_I(\mathcal{A}_x)) = wp(\mathcal{A}_{\langle init \rangle^\frown tr} * I, commgd_I(\mathcal{A}_x))$$

As before, this definition also corresponds to that in CSP.

Theorem 8 *If* $\mathcal{A} = (A, S, \mathcal{A}_i, \mathcal{A}_A, \mathcal{A}_I, dir)$ *then :*

$$\mathcal{A} \text{ deterministic} \Leftrightarrow FDI(\mathcal{A}) \text{ deterministic}$$

Proof As above, the proof follows from the fact that:

$$\overline{wp}(\mathcal{A}_{\langle i \rangle^\frown t^\frown \langle x \rangle} * I, true) \equiv \overline{wp}(\mathcal{A}_{\langle i \rangle^\frown t} * I, gd_I(\mathcal{A}_x))$$

and so

$$\overline{wp}(\mathcal{A}_{\langle i \rangle^\frown t^\frown \langle x \rangle} * I, true) \Rightarrow \neg\ \overline{wp}(\mathcal{A}_{\langle i \rangle^\frown t} * I, \neg\ commgd_I(\mathcal{A}_x))$$
$$\equiv \overline{wp}(\mathcal{A}_{\langle i \rangle^\frown t} * I, gd_I(\mathcal{A}_x)) \Rightarrow wp(\mathcal{A}_{\langle i \rangle^\frown t} * I, commgd_I(\mathcal{A}_x))$$

$\natural$

4.5.4 Other aspects of nondeterminism for action systems

The definitions of action systems given in the previous sections take into account only the internal choice which is reflected in the failures-divergences semantics of an action system. As was noted in Example 18, an action system can make internal choices, but as long as they do not affect the enablement of guards within the system, the failures-divergences representation will be deterministic. It may seem a little strange to regard a system as deterministic when it can make internal choices. However, for the purposes of security this is just what is required, since nondeterminism can compromise security only if its resolution is observable.

The user of an action system is able to observe which events are enabled for them at any stage. If this is altered by internal choices of the system,

then the system is nondeterministic. It is the case that outputs allow a user to view parts of the system state, and it may at first appear that it would be necessary to take additional steps to ensure that high level actions do not update values which will be output to a low level user. However, because of the way outputs are defined, any "writing down" of this sort is reflected by nondeterminism of the system at the interface with the low level user. Hence the nondeterminism conditions will cover this aspect of noninterference too.

Example 24 Consider the following value-passing action system:

$$\mathcal{A}7 \mathrel{\widehat{=}} \left(\begin{array}{l} \textbf{var } x : \{0,1\} \\ \textbf{initially } x := 0 \\ \textbf{action } h \textbf{ in } i? \ :- \mathit{true} \rightarrow x := i? \\ \textbf{action } l \textbf{ out } o! \ :- \mathit{true} \rightarrow o! := x \end{array} \right)$$

where h is a high-level action and l a low-level action. The action for output to l of a particular value, v, is:

$$\begin{aligned} &\mathcal{A}7_{l.v} \equiv (\textbf{local } o! \bullet \mathcal{A}7_l[o! = v]) \\ &\equiv (\textbf{local } o! \bullet \mathit{true} \rightarrow o! := x[o! = v]) \\ &\equiv (\textbf{local } o! \bullet x = v \rightarrow o! := x) \\ &\equiv x = v \rightarrow \mathit{skip} \end{aligned}$$

Hence $\mathcal{A}7_{l.v}$ is only enabled when $x = v$. Whether this is true or not will be a direct consequence of the inputs of h, and so there will be nondeterminism detectable at the low user's interface. ♮

4.6 Summary

This chapter has described the action system notation and showed how action systems are used. A correspondence with CSP was defined for basic action systems and extended to include both systems with internal actions and value-passing systems. Since the failures and divergences of an action system can be calculated, it is possible to apply CSP properties to an action system by obtaining its failures-divergences representation. However, this is not

a very convenient approach and introduces additional layers of complexity which offer additional potential for error. The aim of this thesis is to obtain results which apply directly to action systems. The first step in this process was the definition of a determinism property for action systems which was shown to correspond to determinism in CSP. This paves the way for the work of the following chapter which considers deterministic security properties for action systems.

Chapter 5

Deterministic security for action systems

The two types of deterministic security property, eager and lazy, may be defined directly for action systems without the need to translate the system to CSP failures- divergences. In this chapter the definitions are presented and their equivalence to their CSP counterparts is demonstrated. Given the definition of action system hiding above, eager security for action systems follows straightforwardly. The concepts used to define lazy security do not admit direct translation to action systems without redefinition of the original system and hence we define the lazy condition in a different way and show its equivalence. The use of the action system conditions is shown for some example systems.

5.1 Eager deterministic security

Eager deterministic security in CSP requires a system to be deterministic when all high-level events are hidden. We first consider the case of simple action systems.

5.1.1 Eager security for action systems without value-passing

This definition mirrors that for eager security in CSP.

Definition 24 *If* $\mathcal{A} = (A, S, \mathcal{A}_i, \mathcal{A}_A, \mathcal{A}_I)$ *then* $\mathcal{A}$ *is eagerly secure with the set of actions* $H \subseteq A$ *if* $divs(\mathcal{A} \setminus H) = \emptyset$ *and:*

$$\mathcal{A} \setminus H \quad \textit{deterministic}$$

We check for correspondence with the CSP version.

Theorem 9 *If* $\mathcal{A} = (A, S, \mathcal{A}_i, \mathcal{A}_A, \mathcal{A}_I)$ *then* $\mathcal{A}$ *is eagerly secure with respect to* $H \subseteq A$ *iff* $FDI(\mathcal{A})$ *is eagerly secure with respect to* H*.*

Proof Both the action system hiding operator and the definition of determinism correspond to those in CSP, and this ensures that the eager security property also corresponds. Firstly:

$$\mathcal{A} \setminus H \quad \textit{deterministic} \Leftrightarrow FDI(\mathcal{A} \setminus H) \quad \textit{deterministic} \qquad \text{[Thm 8]}$$

Also, the result of Theorem 1 ensures that:

$$FDI(\mathcal{A} \setminus H) = FDI(\mathcal{A}) \setminus H$$

So:

$$FDI(\mathcal{A} \setminus H) \quad \textit{deterministic} \Leftrightarrow FDI(\mathcal{A}) \setminus H \quad \textit{deterministic}$$

Hence $\mathcal{A}$ is eagerly secure as an action system iff $FDI(\mathcal{A})$ is an eagerly secure process. ♮

5.1.2 Examples

The following examples show how the above definition may be used in practice.

Example 25 First we consider the action system $\mathcal{A}1$ which was defined as:

$$\mathcal{A}1 \mathrel{\widehat{=}} \left(\begin{array}{l} \textbf{var } x : \{0,1\} \\ \textbf{initially } x := 0 \\ \textbf{action } h \ :- \ x = 0 \rightarrow x := 1 \\ \textbf{action } l \ :- \ x = 1 \rightarrow x := 0 \end{array} \right)$$

The system with high-level action h hidden is $\mathcal{A}1 \setminus \{h\}$. For any trace t of visible actions of $\mathcal{A}1 \setminus \{h\}$ we need to determine whether:

$$\overline{wp}(\mathcal{A}1_{\langle i \rangle^\frown t} * \{h\}, gd_{\{h\}}(\mathcal{A}1_l)) = wp(\mathcal{A}1_{\langle i \rangle^\frown t} * \{h\}, gd_{\{h\}}(\mathcal{A}1_l))$$

The postcondition for both simplifies in the following way:

$$\begin{array}{lr}
gd_{\{h\}}(\mathcal{A}1_l) & \\
\equiv \overline{wp}((skip \parallel h); l, true) & \text{[Defn. of } gd \text{ and } \parallel\text{]} \\
\equiv \overline{wp}((l \parallel (h; l), true) & \text{[Prop 2]} \\
\equiv \overline{wp}(l, true) \vee \overline{wp}(h; l, true) & \text{[Defn. } wp \parallel\text{]} \\
\equiv x = 1 \vee x = 0 & \text{[Defn. } \overline{wp}\text{]} \\
\equiv true & \text{[From type of } x\text{]}
\end{array}$$

Using this we have:

$$\begin{array}{lr}
\overline{wp}(\mathcal{A}1_{\langle i \rangle^\frown t} * \{h\}, gd_{\{h\}}(\mathcal{A}1_l)) & \\
\equiv \overline{wp}(\mathcal{A}1_{\langle i \rangle^\frown t} * \{h\}, true) & \\
\equiv true & \text{[Since } t \text{ is a trace]}
\end{array}$$

Also, for the right hand side:

$$\begin{array}{lr}
wp(\mathcal{A}1_{\langle i \rangle^\frown t} * \{h\}, gd_{\{h\}}(\mathcal{A}1_l)) & \\
\equiv wp(\mathcal{A}1_{\langle i \rangle^\frown t} * \{h\}, true) & \\
\equiv true & \text{[Since the system does not diverge]}
\end{array}$$

So $\mathcal{A}1 \setminus \{h\}$ is deterministic and hence $\mathcal{A}1$ is eagerly secure. ♮

Example 26 Here we examine:

$$\mathcal{A}3 \mathrel{\widehat{=}} \left(\begin{array}{l} \textbf{var } x : \{0,1\} \\ \textbf{initially } x := 0 \\ \textbf{action } h \ :- \ x = 0 \rightarrow x :\in \{0,1\} \\ \textbf{action } l \ :- \ x = 1 \rightarrow x := 0 \end{array} \right)$$

It is clear that $\mathcal{A}3 \setminus \{h\}$ diverges since it is possible for h to be executed an infinite number of times. Thus $\mathcal{A}3$ is not eagerly secure. ♮

Example 27 Finally we consider an action system which does not diverge but nevertheless fails to satisfy the eager security property:

$$\mathcal{A}8 \mathrel{\widehat{=}} \left(\begin{array}{l} \textbf{var } x : \{0,1,2\} \\ \textbf{initially } x := 0 \\ \textbf{action } h1 \ :- \ x = 0 \rightarrow x := 1 \\ \textbf{action } h2 \ :- \ x = 0 \rightarrow x := 2 \\ \textbf{action } l1 \ :- \ x = 1 \rightarrow x := 0 \\ \textbf{action } l2 \ :- \ x = 2 \rightarrow x := 0 \end{array} \right)$$

Occurrence of the low-level actions is directly dependent on the high-level actions. $\mathcal{A}8 \setminus \{h1, h2\}$ is certainly not deterministic since, for example, for $l1$:

$$\begin{aligned} & gd_{\{h1,h2\}}(\mathcal{A}8_{l1}) \\ & \equiv \overline{wp}((skip \parallel h1 \parallel h2);\ l1, true) \\ & \equiv \overline{wp}(l1 \parallel (h1;\ l1), true) \\ & \equiv \overline{wp}(l1, true) \vee \overline{wp}((h1;\ l1), true) \\ & \equiv x = 1 \vee x = 0 \end{aligned}$$

Hence:

$$\begin{aligned} & \overline{wp}(\mathcal{A}8_{\langle i \rangle} * \{h1, h2\}, gd_{\{h1,h2\}}(\mathcal{A}8_{l1})) \\ & \equiv \overline{wp}(x := 0;\ IT_{\{h1,h2\}}, x = 0 \vee x = 1) \\ & \equiv \overline{wp}(x := 0;\ x = 0 \vee x = 1) \\ & \equiv true \end{aligned}$$

However:

$$\begin{aligned} & wp(\mathcal{A}8_{\langle i \rangle} * \{h1, h2\}, gd_{\{h1,h2\}}(\mathcal{A}8_{l1})) \\ & \equiv wp(x := 0, wp(IT_{\{h1,h2\}}, x = 0 \vee x = 1)) \\ & \equiv wp(x := 0, wp(skip \parallel h1 \parallel h2, x = 0 \vee x = 1)) \\ & \equiv wp(x := 0, x = 1) \\ & \equiv false \end{aligned}$$

Therefore $\mathcal{A}8$ does not meet the condition for eager security. ♮

5.1.3 Eager security for value-passing action systems

The security condition for value-passing action systems makes use of the *commgd* construct for each visible action:

Definition 25 *If* $\mathcal{A} = (A, S, \mathcal{A}_i, \mathcal{A}_A, \mathcal{A}_I, dir)$ *then* $\mathcal{A}$ *is eagerly secure with the set of actions* $H \subseteq A$ *if* $divs(\mathcal{A} \setminus H) = \emptyset$ *and:*

$\mathcal{A} \setminus H$ *deterministic*

This also corresponds to the CSP version:

Theorem 10 *If* $\mathcal{A} = (A, S, \mathcal{A}_i, \mathcal{A}_A, \mathcal{A}_I, dir)$ *then* $\mathcal{A}$ *is eagerly secure with respect to* $H \subseteq A$ *iff* $FDI(\mathcal{A})$ *is eagerly secure with respect to* H.

Proof As with Theorem 9, the proof here follows since, for value-passing action systems:

$$\mathcal{A} \setminus H \quad \textit{deterministic} \Leftrightarrow FDI(\mathcal{A} \setminus H) \quad \textit{deterministic} \qquad \text{[Thm 8]}$$

and:

$$FDI(\mathcal{A} \setminus H) = FDI(\mathcal{A}) \setminus H \qquad \text{[Thm 1]}$$

$\natural$

5.1.4 Examples

Here, Definition 25 is used to determine whether or not the given value-passing action systems are secure.

Example 28 Action system $\mathcal{A}7$ was defined above as:

$$\mathcal{A}7 \triangleq \left(\begin{array}{l} \textbf{var } x : \{0,1\} \\ \textbf{initially } x := 0 \\ \textbf{action } h \textbf{ in } i? \ :- \ true \rightarrow x := i? \\ \textbf{action } l \textbf{ out } o! \ :- \ true \rightarrow o! := x \end{array} \right)$$

To show that $\mathcal{A}7$ is not eagerly secure, note that $\mathcal{A}7 \setminus \{h\}$ diverges since the guard for h actions is always *true*. For example, the action $h.0$ describing the high-level input of value 0 is calculated from h by:

$$
\begin{aligned}
& \mathcal{A}7_{h.0} \\
& \equiv \mathcal{A}7_h[\textbf{value } 0/i?] && \text{[Defn 13]} \\
& \equiv (\mathit{true} \rightarrow x := i?)[\textbf{value } 0/i?] \\
& \equiv \mathit{true} \rightarrow x := 0
\end{aligned}
$$

and it is clear that an infinite trace of these is possible. ♮

Example 28 demonstrates the fact that no action system which is always ready to accept high-level input can be considered eagerly secure. This is because the possibility of an infinite trace of high-level inputs causes divergence within the system when high-level actions are internalised. This is also true for the CSP security definitions since value-passing for action systems is based on the communication channels of CSP.

Example 29 In this example, a flag is used to indicate when input is possible for h. User h must wait until l has read the value before another is input.

$$
\mathcal{A}9 \mathrel{\widehat{=}} \left(\begin{array}{l}
\textbf{var } x : \{0,1\};\ \mathit{flag} : \{\mathit{on}, \mathit{off}\} \\
\textbf{initially } x := 0;\ \mathit{flag} := \mathit{on} \\
\textbf{action } h \textbf{ in } i? \ :- \mathit{flag} = \mathit{on} \rightarrow x, \mathit{flag} := i?, \mathit{off} \\
\textbf{action } l \textbf{ out } o! \ :- \mathit{flag} = \mathit{off} \rightarrow o!, \mathit{flag} := x, \mathit{on}
\end{array} \right)
$$

This is obviously insecure, and the example shows how the insecurity is revealed by Definition 25. Consider $\mathcal{A}9$ with trace $\langle\rangle$ and action $l.0$. Firstly:

$$
\begin{aligned}
& \mathit{commgd}_{\{h\}}(\mathcal{A}9_{l.0}) \\
& \equiv \overline{wp}(IT_{\{h\}}, \mathit{commgd}(\mathcal{A}9_{l.0})) && \text{[Defn. of } \mathit{commgd}_{\{h\}}\text{]} \\
& \equiv \overline{wp}(IT_{\{h\}}, \mathit{gd}(\mathcal{A}9_l) \wedge \mathit{wp}(\mathcal{A}9_l, o! = 0)) && \text{[Defn. of } \mathit{commgd}\text{]} \\
& \equiv \overline{wp}(IT_{\{h\}}, \mathit{flag} = \mathit{off} \wedge x = 0) \\
& \equiv \overline{wp}(\mathit{skip} \,[\!]\, h.0, \mathit{flag} = \mathit{off} \wedge x = 0)
\end{aligned}
$$

$$\equiv \overline{wp}(skip, flag = off \wedge x = 0) \vee \overline{wp}(h.0, flag = off \wedge x = 0)$$
$$\equiv (flag = off \wedge x = 0) \vee flag = on$$

Similarly:

$$gd_{\{h\}}(\mathcal{A}9_{l.0}) \equiv (flag = off \wedge x = 0) \vee flag = on$$

Using this:

$$\overline{wp}(\mathcal{A}9_{\langle i \rangle} * \{h\}, commgd_{\{h\}}(\mathcal{A}9_{l.0}))$$
$$\equiv \overline{wp}(x := 0;\ flag := on;\ IT_{\{h\}}, (flag = off \wedge x = 0) \vee flag = on)$$
$$\equiv true$$

Also:

$$wp(\mathcal{A}9_{\langle i \rangle} * \{h\}, commgd_{\{h\}}(\mathcal{A}9_{l.0}))$$
$$\equiv wp(x := 0;\ flag := on;\ IT_{\{h\}}, (flag = off \wedge x = 0) \vee flag = on)$$
$$\equiv wp(x := 0;\ flag := on,$$
$$\qquad wp(IT_{\{h\}}, (flag = off \wedge x = 0) \vee flag = on))$$
$$\equiv wp(x := 0;\ flag := on,$$
$$\qquad wp(skip \parallel h.0 \parallel h.1, (flag = off \wedge x = 0) \vee flag = on)))$$
$$\equiv wp(x := 0;\ flag := on, flag = off \wedge x = 0)$$
$$\equiv false$$

Hence the system with high-level actions hidden is nondeterministic. ♮

Example 30 Although low-level actions may not depend on high-level actions, it is permissible for high-level actions to be influenced by low-level ones. This situation occurs in the following system:

$$\mathcal{A}10 \triangleq \left(\begin{array}{l} \textbf{var } x : \{0,1\}, flag : \{on, off\} \\ \textbf{initially } x := 0;\ flag := on \\ \textbf{action } l \textbf{ in } i? \ :- flag = on \rightarrow x, flag := i?, off \\ \textbf{action } h \textbf{ out } o! \ :- flag = off \rightarrow o!, flag := x, on \end{array} \right)$$

For any value $v \in \{0,1\}$:

$$
\begin{aligned}
& commgd_{\{h\}}(\mathcal{A}10_{l.v}) \\
& \equiv \overline{wp}(IT_{\{h\}}, commgd(\mathcal{A}10_{l.v})) \\
& \equiv \overline{wp}(IT_{\{h\}}, flag = on) && \text{[Defn. of } commgd \text{ for input]} \\
& \equiv true
\end{aligned}
$$

Similarly:

$$gd_{\{h\}}(\mathcal{A}10_{l.v}) \equiv true$$

For any trace, t, of $\mathcal{A}10 \setminus \{h\}$:

$$
\begin{aligned}
& \overline{wp}(\mathcal{A}10_{\langle i \rangle^\frown t} * \{h\}, true) \\
& \equiv true && [t \text{ is a trace}]
\end{aligned}
$$

and:

$$
\begin{aligned}
& wp(\mathcal{A}10_{\langle i \rangle^\frown t} * \{h\}, true) \\
& \equiv true && [\mathcal{A}10 \text{ does not diverge}]
\end{aligned}
$$

Hence $\mathcal{A}10$ is secure with respect to Definition 25 ♭

5.1.5 Action systems and lazy deterministic security

The definition of lazy deterministic security does not translate so easily from CSP. This is because the nature of action systems places limitations on the operations which can be applied. The CSP definition of lazy security uses RUN_H and interleaving, both of which can be defined for an action system only by redefinition of the original actions. For example:

$$
\mathcal{A}1 \mathrel{\widehat{=}} \left(\begin{array}{l}
\textbf{var } x : \{0,1\} \\
\textbf{initially } x := 0 \\
\textbf{action } h \ :- \ x = 0 \rightarrow x := 1 \\
\textbf{action } l \ :- \ x = 1 \rightarrow x := 0
\end{array} \right)
$$

is equivalent to:

$$P = h \rightarrow l \rightarrow P$$

In CSP, $RUN_{\{h,l\}}$ consists of all possible traces of h and l. In action systems the action definitions fix the ordering in which events can occur and so some sequences can never be a trace of the system. For instance, the sequence $\langle h, h \rangle$ cannot be a trace since:

$$\overline{wp}(x := 0;\ x = 0 \rightarrow x := 1;\ x = 0 \rightarrow x := 1, \mathit{true})$$
$$= \mathit{false}$$

It is certainly possible to construct an action system equivalent to RUN_H, for example:

$$\left(\begin{array}{l} \textbf{var} \\ \textbf{initially } \mathit{skip} \\ \textbf{action } h \ :- \ \mathit{skip} \end{array}\right)$$

but the action h must be redefined.

Similarly, interleaving in CSP allows actions from a given set to appear anywhere in a trace, which may not be possible in an action system. In general, constructs such as these cannot be defined for action systems without redefining the actions themselves. In the following sections we suggest a simple way to amend the definition of H actions, which has the same effect (in terms of the CSP semantics) as interleaving with RUN_H. We first consider action systems without value-passing or internal actions.

5.1.6 Lazy deterministic security for simple action systems

In order to define lazy deterministic security for action systems we take a different approach. We construct an action system whose failures and divergences are equivalent to those of the equivalent CSP process interleaved with RUN_H. First we identify what those failures and divergences should be.

The process RUN_H has all traces of elements of H. It can never refuse to engage in an H event and can never engage in any non-H event. RUN_H is also divergence-free. Hence the divergences of $P \,|||\, RUN_H$ are:

$$\{u \hat{} w \mid w \in (\alpha P \cup H)^* \wedge \\ (\exists s \in traces(P);\ t \in traces(RUN_H) \bullet s \in divs(P) \wedge u \in s \,|||\, t)\}$$

That is, a divergence of $P \,|||\, RUN_H$ is a divergence of P with arbitrary elements of H interspersed.

The failures of $P \,|||\, RUN_H$ include all (s, X) for any divergence s, plus:

$$\{(v, (X \cap Y)) \mid \exists (s, X) \in fails(P);\ (t, Y) \in fails(RUN_H) \bullet \\ v \in s \,|||\, t\}$$

A failure of RUN_H consists of any trace of H events and any set (including $\emptyset$) of L events. So the failures of $P \,|||\, RUN_H$ are the failures of P with each trace interspersed with H events and all H events removed from each refusal set.

To create an action system which obscures H events in the same way as $P \,|||\, RUN_H$ does, H events must be allowed to occur at any point during execution of the system. This suggests that weakening the guard to *true* may help. However, the command part of each action must also be altered to prevent "interleaved" occurrences of H events from introducing unwanted divergence. This leads to the following definition:

Definition 26 *If $\mathcal{A}$ is the action system $(A, S, \mathcal{A}_i, \mathcal{A}_A, \mathcal{A}_I, dir)$ and H a set of actions of $\mathcal{A}$ then $obs_H(\mathcal{A})$ is the action system obtained from $\mathcal{A}$ by obscuring each $h \in H$ in the following way:*

$$obs_H(\mathcal{A})_h \mathrel{\hat{=}} \mathcal{A}_h \,[\!]\, (true \rightarrow skip) \qquad for\ h \in H$$
$$obs_H(\mathcal{A})_a \mathrel{\hat{=}} \mathcal{A}_a \qquad for\ a \in A - H$$

The effect of H actions is obscured by allowing each one to make an internal decision to *skip*. Hence H actions may occur at any point during the execution of $obs_H(\mathcal{A})$. Each high-level action h makes a nondeterministic

choice between skipping or, within the guard of $\mathcal{A}_h$, behaving as $\mathcal{A}_h$. It follows from the definition that:

$$\overline{wp}(obs_H(\mathcal{A})_h, \alpha) = \overline{wp}(\mathcal{A}_h, \alpha) \vee \alpha$$

Using obs_H we can define lazy security for action systems, since $\mathcal{A}$ will be lazily secure if $obs_H(\mathcal{A})$ is deterministic.

Definition 27 *If $\mathcal{A} \mathrel{\widehat{=}} (A, S, \mathcal{A}_i, \mathcal{A}_A)$ then $\mathcal{A}$ is lazily secure for the set of actions $H \subseteq A$ if for all $tr \in traces(obs_H(\mathcal{A}))$ and for all $x \in A \setminus H$:*

$$\overline{wp}(obs_H(\mathcal{A})_{\langle i \rangle^\frown tr}, gd(\mathcal{A}_x)) = wp(obs_H(\mathcal{A})_{\langle i \rangle^\frown tr}, gd(\mathcal{A}_x))$$

5.1.7 Correspondence with CSP

To demonstrate that the representation of obscuring is satisfactory we need to show:

Theorem 11 *For action system $\mathcal{A} = (A, S, \mathcal{A}_i, \mathcal{A}_A)$ and $H \subseteq A$:*

$$FD(obs_H(\mathcal{A})) =_{FD} FD(\mathcal{A}) \,|||\, RUN_H$$

Proof Any H in $obs_H(\mathcal{A})$ may choose to skip. So if t is a trace of $\mathcal{A}$ such that for postcondition ϕ:

$$\overline{wp}(\mathcal{A}_{\langle i \rangle^\frown t}, \phi)$$

then for any $s \in H^*$ and any $int \in t \,|||\, s$:

$$\overline{wp}(\mathcal{A}_{\langle i \rangle^\frown int}, \phi)$$

A divergence of $FD(\mathcal{A}) \,|||\, RUN_H$ is any divergence of $FD(\mathcal{A})$ interspersed with elements of H. For $obs_H(\mathcal{A})$, if t is a divergence of $\mathcal{A}$ then:

$$\overline{wp}(\mathcal{A}_{\langle i \rangle^\frown t}, false)$$

So from the fact noted above, for any $s \in H^*$ and any $int \in t \,|||\, s$ we have:

$$\overline{wp}(\mathcal{A}_{\langle i \rangle^\frown int}, false)$$

Hence any divergence of $\mathcal{A}$ interspersed with elements of H is a divergence of $obs_H(\mathcal{A})$.

No new divergences can be introduced since the only new possibility for the altered actions is the option to skip. Hence the divergences of $obs_H(\mathcal{A})$ and those of $FD(\mathcal{A}) \mid\mid\mid RUN_H$ are equal.

A failure of $FD(\mathcal{A}) \mid\mid\mid RUN_H$ is a failure, (t, X), of $FD(\mathcal{A})$ with t interspersed with elements of H and H events removed from X. For $\mathcal{A}$, if (t, X) is a failure then:

$$\overline{wp}(\mathcal{A}_{\langle i \rangle^\frown t}, \neg\ gd(\mathcal{A}_X))$$

In $obs_H(\mathcal{A})$ the guards of H actions are weakened to *true*, so H events are always enabled. However, the subset of X containing all non-H actions will still be included in the failure pair, so:

$$\overline{wp}(obs_H(\mathcal{A})_{\langle i \rangle^\frown t}, \neg\ gd(obs_H(\mathcal{A})_{X \setminus H}))$$

and hence $(t, X \setminus H)$ is a failure of $obs_H(\mathcal{A})$. Again, from the fact noted above, it follows that for any $s \in H^*$ and any $int \in t \mid\mid\mid s$:

$$\overline{wp}(obs_H(\mathcal{A})_{\langle i \rangle^\frown int}, \neg\ gd(obs_H(\mathcal{A}_{X \setminus H})))$$

and so $(int, X \setminus H)$ is also a failure of $obs_H(\mathcal{A})$.

The only additional behaviour of $obs_H(\mathcal{A})$ is the ability of H actions to skip, so no new failures can be introduced. Hence the failures of $obs_H(\mathcal{A})$ are equal to those of $FD(\mathcal{A}) \mid\mid\mid RUN_H$. ♮

5.1.8 Examples

The following examples show how Definition 27 is applied.

Example 31 As before we have:

$$\mathcal{A}1 \mathrel{\widehat{=}} \left(\begin{array}{l} \textbf{var } x : \{0,1\} \\ \textbf{initially } x := 0 \\ \textbf{action } h \ :- \ x = 0 \rightarrow x := 1 \\ \textbf{action } l \ :- \ x = 1 \rightarrow x := 0 \end{array} \right)$$

For lazy deterministic security we need to check whether:

$$obs_H(\mathcal{A}1) \mathrel{\widehat{=}} \left(\begin{array}{l} \textbf{var } x : \{0,1\} \\ \textbf{initially } x := 0 \\ \textbf{action } h \ :-(x = 0 \rightarrow x := 1) \,[\!]\, (true \rightarrow skip) \\ \textbf{action } l \ :- \ x = 1 \rightarrow x := 0 \end{array} \right)$$

is deterministic. However:

$$\overline{wp}(obs_H(\mathcal{A}1)_{\langle i;\, h\rangle}, gd(l)) = true$$

but

$$wp(obs_H(\mathcal{A}1)_{\langle i;\, h\rangle}, gd(l)) = false$$

and so, as we would expect, $\mathcal{A}1$ is not lazily secure.

Example 32

$$\mathcal{A}6 \mathrel{\widehat{=}} \left(\begin{array}{l} \textbf{var } x : \{0,1\} \\ \textbf{initially } x := 0 \\ \textbf{action } h \ :- \ true \rightarrow x := 1 \\ \textbf{action } l \ :- \ true \rightarrow x := 0 \end{array} \right)$$

$\mathcal{A}6$ is not eagerly secure because hiding h leads to divergence. Obscuring H events in $\mathcal{A}6$ gives:

$$obs_H(\mathcal{A}6) \mathrel{\widehat{=}} \left(\begin{array}{l} \textbf{var } x : \{0,1\} \\ \textbf{initially } x := 0 \\ \textbf{action } h \ :-(true \rightarrow x := 1) \,[\!]\, skip \\ \textbf{action } l \ :- \ true \rightarrow x := 0 \end{array} \right)$$

The guards of both actions are always enabled and so $obs_H(\mathcal{A}6)$ is deterministic. Hence $\mathcal{A}6$ is lazily secure.

5.1.9 Lazy deterministic security and internal events

Definition 26 is equally applicable to action systems with internal actions this is confirmed by the following theorem.

Theorem 12 *For action system* $\mathcal{A} \mathrel{\widehat{=}} (A, S, \mathcal{A}_i, \mathcal{A}_A, \mathcal{A}_I)$ *and* $H \subseteq A$*:*

$$FDI(obs_H(\mathcal{A})) = FDI(\mathcal{A}) \,|||\, RUN_H$$

Proof The equivalence for failures and divergences follows in the same way as in Theorem 11. An infinite trace of $FDI(\mathcal{A}) \,|||\, RUN_H$ is either an infinite trace of $FDI(\mathcal{A})$ interspersed with elements of H, or it is an infinite sequence of H events (possibly both). This is also the case for $obs_H(\mathcal{A})$, so the infinite traces are equal. ♮

So for action systems with internal actions:

Definition 28 *If* $\mathcal{A} \mathrel{\widehat{=}} (A, S, \mathcal{A}_i, \mathcal{A}_A, \mathcal{A}_I)$ *then* $\mathcal{A}$ *is lazily secure for the set of actions* $H \subseteq A$ *if for all* $tr \in traces(obs_H(\mathcal{A}))$ *and for all* $x \in A \setminus H$*:*

$$\overline{wp}(obs_H(\mathcal{A})_{\langle i \rangle^\frown tr} * I, gd_I(\mathcal{A}_x)) = wp(obs_H(\mathcal{A})_{\langle i \rangle^\frown tr} * I, gd_I(\mathcal{A}_x))$$

5.1.10 Lazy deterministic security for value-passing action systems

The situation is also similar for value-passing action systems. The $obs_H(\mathcal{A})$ action for any $h \in H$ has been defined:

$$obs_H(\mathcal{A})_h = \mathcal{A}_h \,\|\, skip$$

We can therefore calculate input and output actions for specific values as follows. If $dir(h) = in$ then:

$$\begin{aligned}
&obs_H(\mathcal{A})_{h.v} \\
&\equiv (\mathcal{A}_h \,\|\, skip)[\textbf{value } v/i?] \\
&\equiv ((\mathcal{A}_h)[\textbf{value } v/i?]) \,\|\, skip \\
&\equiv \mathcal{A}_{h.v} \,\|\, skip
\end{aligned}$$

If $dir(h) = out$ then:

$$\begin{aligned}
&obs_H(\mathcal{A})_{h.v} \\
&\equiv (\textbf{local } o! \bullet (\mathcal{A}_h \,\|\, skip)[o! = v]) \\
&\equiv (\textbf{local } o! \bullet (\mathcal{A}_h)[o! = v]) \,\|\, skip) \\
&\equiv \mathcal{A}_{h.v} \,\|\, skip
\end{aligned}$$

Considering the case of an obscured output action, the guard of $h.v$ is always *true* (because the option to skip is always available) and so the output of any particular value is always enabled. Effectively, the user is offered the choice between all possible output values on that channel. This accords with the effect of obscuring in CSP where an obscured output channel offers the choice of value externally and hence can be viewed as an input channel. In fact, it does not really matter how we refer to the channel as long as the failures-divergences semantics agrees with the CSP equivalent. The following theorem shows that this consistency with CSP does indeed hold.

Theorem 13 *For action system* $\mathcal{A} \mathrel{\widehat{=}} (A, S, \mathcal{A}_i, \mathcal{A}_A, \mathcal{A}_I, dir)$ *and* $H \subseteq A$*:*

$$FDI(obs_H(\mathcal{A})) = FDI(\mathcal{A}) \;|||\; RUN_H$$

Essentially, the only difference is that *commgd* is used in the definition of failures and the proof follows as before. Once again we have:

Definition 29 *If* $\mathcal{A} \mathrel{\widehat{=}} (A, S, \mathcal{A}_i, \mathcal{A}_A, \mathcal{A}_I, dir)$ *then* $\mathcal{A}$ *is lazily secure for the set of actions* $H \subseteq A$ *if for all* $tr \in traces(obs_H(\mathcal{A}))$ *and for all* $x \in A \setminus H$*:*

$$\overline{wp}(obs_H(\mathcal{A})_{\langle i \rangle^\frown tr} * I, gd_I(\mathcal{A}_x)) = wp(obs_H(\mathcal{A})_{\langle i \rangle^\frown tr} * I, commgd_I(\mathcal{A}_x))$$

5.1.11 Examples

In the following examples Definition 29 is used to establish whether or not the given value-passing action systems are lazily secure.

Example 33 Once again we consider action system $\mathcal{A}7$ of Example 24:

$$\mathcal{A}7 \mathrel{\widehat{=}} \left(\begin{array}{l} \textbf{var } x : \{0,1\} \\ \textbf{initially } x := 0 \\ \textbf{action } h \textbf{ in } i? \;:- \; true \rightarrow x := i? \\ \textbf{action } l \textbf{ out } o! \;:- \; true \rightarrow o! := x \end{array} \right)$$

This failed to be eagerly secure since hiding high-level actions causes divergence. It is also insecure with respect to the lazy definition. We confirm this

by looking at the system with H actions obscured:

$$obs_h(\mathcal{A}7) \mathrel{\widehat{=}} \left(\begin{array}{l} \textbf{var } x : \{0,1\} \\ \textbf{initially } x := 0 \\ \textbf{action } h \textbf{ in } i? \ :- \ (true \rightarrow x := i?) \,[\!]\, (true \rightarrow skip) \\ \textbf{action } l \textbf{ out } o! \ :- \ true \rightarrow o! := x \end{array} \right)$$

This is nondeterministic since for $v \in \{0,1\}$ the action $h.v$ can either enable $l.v$ or it can skip. ♮

Example 34 Here we look at system $\mathcal{A}5$ defined in Example 22. We will let $H = \{take\_off\}$ and $L = \{put\_on\}$. This gives:

$$obs_H(\mathcal{A}5) \mathrel{\widehat{=}} \left(\begin{array}{l} \textbf{var } s : \text{seq } \mathbb{N} \\ \textbf{initially } s := \langle\rangle \\ \textbf{action } put\_on \textbf{ in } i? : \mathbb{N} \ :- \\ \quad true \rightarrow s := s^\frown\langle i?\rangle \\ \textbf{action } take\_off \textbf{ out } o! : \mathbb{N} \ :- \\ \quad (s \neq \langle\rangle \rightarrow s, o! := front\ s, last\ s) \,[\!]\, (true \rightarrow skip) \end{array} \right)$$

For any low-level action, $put\_on.v$:

$$commgd(obs_H(\mathcal{A}5)_{put\_on.v}) = true$$

and for any trace t:

$$\overline{wp}(obs_H(\mathcal{A}5)_{\langle i\rangle^\frown t}, true) = wp(obs_H(\mathcal{A}5)_{\langle i\rangle^\frown t}, true) = true$$

Since high-level actions have the option to skip they are always enabled. Therefore, in investigating the nondeterminism of a system we have to consider the enablement of low-level actions only. Hence $\mathcal{A}5$ with H as defined here is lazily secure. ♮

In the previous example it was a high-level output action which was obscured. The effect of this is to enable the output of any value required by the user at any point in the execution of the system. That is, *take_off* is now prepared to communicate whatever value the environment wishes. Following

the CSP interpretation, since the value communication is now under the control of the environment, this is equivalent to an input channel. The action is prepared to communicate any value chosen by the environment, but only in the case of that value being equivalent to *last* s can the action choose to diminish s (although even in this case it can choose not to). Hence another (and perhaps more satisfactory) way of viewing the obscured action would be as an input:

$$\textbf{action } take\_off \textbf{ in } o? : \mathbb{N} \ :-$$
$$(s \neq \langle\rangle \wedge o? = last\ s \rightarrow s := front\ s) \,[\!]\, (true \rightarrow skip)$$

However, the way in which the action is viewed is really not important for present purposes, since the action is not part of a system to be implemented, but merely a device in the definition of the security property. The important point is that these definitions are equivalent in terms of their behaviour in the definition, and this follows since for both cases the action to communicate any value v is calculated as:

$$take\_off.v \equiv (s \neq \langle\rangle \wedge v = last\ s \rightarrow s := front\ s) \,[\!]\, (true \rightarrow skip)$$

5.2 Mixed security conditions

As was described in Chapter 3, in the CSP approach of Roscoe [107] a mixed security condition combining both eager and lazy definitions is used. This allows signal events which cannot delay the system to be viewed in an eager fashion whilst the rest of the system must maintain lazy security.

5.2.1 Mixed security for action systems

A similar definition may be given for action systems.

Definition 30 *If* $\mathcal{A} \mathrel{\widehat{=}} (A, S, \mathcal{A}_i, \mathcal{A}_A, \mathcal{A}_I, dir)$ *where the high- level actions* $H \subseteq A$ *are divided into delay events,* D*, and signal events,* S*, then* $\mathcal{A}$ *satisfies the mixed security condition if:*

$obs_D(\mathcal{A} \setminus S)$ *deterministic*

Example 35 The following action system is a suitable candidate for consideration under the mixed security condition.

$$\mathcal{A}11 \mathrel{\widehat{=}} \left(\begin{array}{l} \textbf{var } \mathit{flag} : \{\mathit{on}, \mathit{off}\} \\ \textbf{initially } \mathit{flag} := \mathit{off} \\ \textbf{action } h1 \ :- \ \mathit{flag} = \mathit{off} \rightarrow \mathit{flag} := \mathit{on} \\ \textbf{action } h2 \ :- \ \mathit{flag} = \mathit{on} \rightarrow \mathit{flag} := \mathit{off} \\ \textbf{action } l \ :- \ \mathit{flag} = \mathit{off} \rightarrow \mathit{skip} \end{array} \right)$$

In $\mathcal{A}11$, l can only occur when the flag is *off*. The high-level action $h1$ may also occur when the flag is *off*, and this action resets the flag to *on*. Action l cannot then occur until $h2$ has set the flag back to *off*. That this fails to meet the lazy security condition can be seen by considering:

$$obs_H(\mathcal{A}11) \mathrel{\widehat{=}} \left(\begin{array}{l} \textbf{var } \mathit{flag} : \{\mathit{on}, \mathit{off}\} \\ \textbf{initially } \mathit{flag} := \mathit{off} \\ \textbf{action } h1 \ :- \ (\mathit{flag} = \mathit{off} \rightarrow \mathit{flag} := \mathit{on}) \\ \qquad\qquad\qquad \| \ (\mathit{true} \rightarrow \mathit{skip}) \\ \textbf{action } h2 \ :- \ (\mathit{flag} = \mathit{on} \rightarrow \mathit{flag} := \mathit{off}) \\ \qquad\qquad\qquad \| \ (\mathit{true} \rightarrow \mathit{skip}) \\ \textbf{action } l \ :- \ \mathit{flag} = \mathit{off} \rightarrow \mathit{skip} \end{array} \right)$$

Here, $\langle h_1, h_2 \rangle$ may enable l or it may not, so $obs_H(\mathcal{A}11)$ is nondeterministic. However, we could choose to regard $h2$ as a signal event, resetting the flag to *off* as soon as it is able to. In this case we should check $\mathcal{A}11$ against the mixed security condition. Here, the guard of the low-level action in $obs_D(\mathcal{A}11) \setminus S$ is:

$$\begin{aligned} & gd_{\{h2\}}(\mathit{flag} = \mathit{off} \rightarrow \mathit{skip}) \\ & \equiv gd(IT_{\{h2\}};\ (\mathit{flag} = \mathit{off} \rightarrow \mathit{skip})) \\ & \equiv gd(\mathit{skip} \ \| \ h2;\ (\mathit{flag} = \mathit{off} \rightarrow \mathit{skip})) \\ & \equiv (\mathit{flag} = \mathit{off}) \vee (\mathit{flag} = \mathit{on}) \\ & \equiv \mathit{true} \end{aligned}$$

The system does not diverge, so it is not possible for any trace of events to affect whether l is enabled or not (it always is). So $obs_D(\mathcal{A}11) \setminus S$ is deterministic and hence $\mathcal{A}11$ meets the mixed security condition. ♮

5.3 Abstract models of high-level behaviour

In Section 3.3 an alternative definition of the deterministic security properties was given. This defined in each case the most nondeterministic behaviour possible for the high-level user. The condition which then must be checked is of the form:

$$(P \parallel_H U_H) \setminus H \ det$$

In the following sections we discuss how this abstraction may be expressed for action systems.

5.3.1 High-level behaviour

HAVOC and *FINITE* are the two basic behaviours needed for these definitions. *FINITE* simply introduces a bound on the possible number of iterations of H actions.

$$FINITE_H \mathrel{\widehat{=}} \left(\begin{array}{l} \textbf{var } count : \mathbb{N} \\ \textbf{initially } count :\in \mathbb{N} \\ \textbf{action } h_i \in H \ :- \ count > 0 \rightarrow count := count - 1 \end{array} \right)$$

$FINITE_H$ has as traces all sequences of H actions but, on execution, repetition will always be bounded by the initial choice of *count*. $FINITE_H$ has no divergences.

To define *HAVOC* we use a variable *select* which restricts the choice of the next action to be executed. If *select* is the emptyset then no further H actions can occur. Using this we have:

$$HAVOC_H \mathrel{\widehat{=}} \left(\begin{array}{l} \textbf{var } select : \mathbb{P}\ H \\ \textbf{initially } select :\subseteq H \\ \textbf{action } h_i \in H \ :- \ h_i \in select \rightarrow select :\subseteq H \end{array} \right)$$

Unlike $FINITE_H$, infinite traces of H actions can occur in $HAVOC_H$ but their failures and divergences are equivalent. The definition of parallel composition can be used to give security definitions in terms of the independent

characterisation of high-level behaviour. These security definitions are given in the following sections. It is intended that the variables *count* and *select* should be different from the variables of any action system with which they are composed. If a clash of variables occurs then different variables can be chosen for use in *FINITE* and *HAVOC*.

5.3.2 Strong deterministic security

A system which exhibits both lazy and eager security is strongly secure (see page 48). The abstract behaviour U_H is in this case described by $HAVOC_H$. Hence the condition to be checked is:

$$(P \underset{H}{\|} HAVOC_H) \setminus H \quad \textit{deterministic}$$

Example 36 For $\mathcal{A}1$ defined in Example 16, $(\mathcal{A} \underset{H}{\|} HAVOC_H) \setminus H$ is:

$$\left(\begin{array}{l} \textbf{var } x : \{0,1\};\ \textit{select} : \mathbb{P}\ \{h\} \\ \textbf{initially } x := 0;\ \textit{select} :\subseteq \{h\} \\ \textbf{action } l\ :- x = 1 \rightarrow x := 0 \\ \textbf{internal } h\ :- (x = 0 \wedge h \in \textit{select}) \rightarrow \\ \qquad\qquad\qquad (x := 1;\ \textit{select} :\subseteq \{h\}) \end{array}\right)$$

Here, for example, $\langle l \rangle$ is a trace since the initialisation may set *select* to $\{h\}$, thus enabling the internal action. However, in the case that *select* is set to $\emptyset$ initially, the system is immediately deadlocked. Thus, as expected, $\mathcal{A}1$ is not strongly secure. ♮

5.3.3 Eager deterministic security

As pointed out in [107], this case is trivial. In CSP the appropriate high-level user is characterised by RUN_H. Since $(P \underset{H}{\|} RUN_H) = P$ for all P and H, the abstract high-level user approach is equivalent to the original definition of eager security:

$$((P \underset{H}{\|} RUN_H) \setminus H) \quad \textit{deterministic} \Leftrightarrow (P \setminus H) \quad \textit{deterministic}$$

5.3.4 Lazy deterministic security

To define appropriate high-level user behaviour for the case of lazy security, Roscoe [107] uses the infinite traces model of CSP. The high-level user's behaviour is described by the process $FINITE_H$. Hence a system is lazily deterministic if:

$$(P \underset{H}{\|} FINITE_H) \setminus H \quad \textit{deterministic}$$

Example 37 The following action system is not eagerly secure since hiding h leads to divergence.

$$\mathcal{A}13 \mathrel{\widehat{=}} \left(\begin{array}{l} \textbf{var } x : \{0,1\} \\ \textbf{initially } x :\in \{0,1\} \\ \textbf{action } h \ :- \ \textit{true} \rightarrow x := 0 \\ \textbf{action } l \ :- \ \textit{true} \rightarrow x := 1 \end{array} \right)$$

This example shows how this approach deals with the possibility of divergence amongst hidden H actions. The system representing $\mathcal{A}13$ with the most nondeterministic characterisation of H compatible with lazy security is given by $(\mathcal{A}13 \underset{H}{\|} FINITE_H) \setminus H$, which is:

$$\left(\begin{array}{l} \textbf{var } x : \{0,1\};\ \textit{count} : \mathbb{N} \\ \textbf{initially } x :\in \{0,1\};\ \textit{count} :\in \mathbb{N} \\ \textbf{action } l \ :- \ \textit{true} \rightarrow x := 1 \\ \textbf{internal } h \ :- \ \textit{count} > 0 \rightarrow x, \textit{count} := 0, \textit{count} - 1 \end{array} \right)$$

Hiding h does not cause divergence since *count* acts as an upper bound on the number of occurences of h. Further, $(\mathcal{A}13 \underset{H}{\|} FINITE_H) \setminus H$ is deterministic (since the guard of l is *true* and the system does not diverge) and so $\mathcal{A}13$ is lazily secure. ♮

Example 38 To see how systems which are not lazily secure fail to meet the requirement we use system $\mathcal{A}1$ again. We construct $(\mathcal{A}1 \underset{H}{\|} FINITE_H) \setminus H$:

$$\left(\begin{array}{l} \textbf{var } x : \{0,1\};\ \textit{count} : \mathbb{N} \\ \textbf{initially } x := 0;\ \textit{count} :\in \mathbb{N} \\ \textbf{action } l \ :- \ x = 1 \rightarrow x := 0 \\ \textbf{internal } h \ :- \ (x = 0 \wedge \textit{count} > 0) \rightarrow x, \textit{count} := 1, \textit{count} - 1 \end{array} \right)$$

In fact, *count* has little effect here, since the system is nondeterministic because of the possible values of x after IT_H. ♮

5.3.5 Mixed deterministic security

In CSP mixed deterministic security is defined for a process where high-level events are divided into two classes: delay events, D, and signal events, S. The signal events are thought of as happening instantaneously whenever possible, such as would be the case with a high-level output sent by the system in response to some high- level input. The CSP mixed security condition is:

$$((P \underset{H}{\|} (RUN_S \;|||\; FINITE_D)) \setminus H) \quad \textit{deterministic}$$

where S, D partition H.

Since S and D are disjoint the action system equivalent of $RUN_S \;|||\; FINITE_D$ can be obtained simply by putting together the actions from both:

$$\left(\begin{array}{l} \textbf{var } count : \mathbb{N} \\ \textbf{initially } count :\in \mathbb{N} \\ \textbf{action } h_s \in S \;:- \; true \rightarrow skip \\ \textbf{action } h_d \in D \;:- \; count > 0 \rightarrow count := count - 1 \end{array}\right)$$

Example 39

$$\mathcal{A}14 \mathrel{\widehat{=}} \left(\begin{array}{l} \textbf{var } x : \{0,1\} \\ \textbf{initially } x := 0 \\ \textbf{action } h1 \;:- \; true \rightarrow skip \\ \textbf{action } h2 \;:- \; true \rightarrow x := 1 \\ \textbf{action } l \;:- \; x = 0 \rightarrow skip \end{array}\right)$$

Here we consider $h1$ as the delay event and $h2$ as the signal event. In order to assess $\mathcal{A}14$ under the mixed security condition we construct $(\mathcal{A}14 \underset{H}{\|} (RUN_{\{h2\}} \;|||\; FINITE_{\{h1\}})) \setminus H$:

$$\left(\begin{array}{l} \textbf{var } x : \{0,1\};\; count : \mathbb{N} \\ \textbf{initially } x := 0;\; count :\in \mathbb{N} \\ \textbf{action } l \;:- \; x = 0 \rightarrow skip \\ \textbf{internal } h1 \;:- \; count > 0 \rightarrow count := count - 1 \\ \textbf{internal } h2 \;:- \; true \rightarrow x := 1 \end{array}\right)$$

However, this diverges with unbounded repetition of the internal action $h2$, so $\mathcal{A}14$ is not secure with respect to the mixed security condition. ♮

Example 40 By altering the actions of h we can produce a system which is secure:

$$\mathcal{A}15 \mathrel{\hat{=}} \left(\begin{array}{l} \textbf{var } x : \{0,1\} \\ \textbf{initially } x := 0 \\ \textbf{action } h1 \ :- \ x = 0 \rightarrow x := 1 \\ \textbf{action } h2 \ :- \ x = 1 \rightarrow x := 0 \\ \textbf{action } l \ :- \ x = 0 \rightarrow \textit{skip} \end{array} \right)$$

Again, for the mixed security condition with $h1$ the delay action and $h2$ the signal action we consider $(\mathcal{A}15 \parallel_H (RUN_{\{h2\}} \ ||| \ FINITE_{\{h1\}})) \setminus H$:

$$\left(\begin{array}{l} \textbf{var } x : \{0,1\};\ \textit{count} : \mathbb{N} \\ \textbf{initially } x := 0;\ \textit{count} :\in \mathbb{N} \\ \textbf{action } l \ :- \ x = 0 \rightarrow \textit{skip} \\ \textbf{internal } h1 \ :- \ (x = 0 \wedge \textit{count} > 0) \rightarrow x, \textit{count} := 1, \textit{count} - 1 \\ \textbf{internal } h2 \ :- \ x = 1 \rightarrow x := 0 \end{array} \right)$$

We need to check whether this is deterministic. In fact, with the high level actions hidden the guard of the low-level action becomes:

$$\begin{aligned} & gd_{\{h1,h2\}}(l) \\ & \equiv gd(IT_{\{h1,h2\}}, x = 0) \\ & \equiv x = 0 \vee x = 1 \\ & \equiv \textit{true} \end{aligned}$$

Thus l is always enabled. So $(\mathcal{A}15 \parallel_H (RUN_{\{h2\}} \ ||| \ FINITE_{\{h1\}})) \setminus H$ is deterministic and $\mathcal{A}15$ satisfies the mixed security condition. ♮

5.3.6 Conditional security

Conditional security, as introduced in [43], specifies that a system is secure dependent upon some condition. For Example 15 of Chapter 3, the managerial tasks, M, of the high-level user may breach the security policy. The user

has to be trusted to perform these judiciously. The security condition says that all other behaviours within the system must be lazily secure:

$$(P \underset{H}{\|} FINITE_{H-M}) \setminus H \quad \textit{deterministic}$$

The same can be done for action systems using the corresponding action system, $FINITE_{H-M}$.

Example 41

$$\mathcal{A}14 \mathrel{\widehat{=}} \left(\begin{array}{l} \textbf{var } x : \{0,1\} \\ \textbf{initially } x := 0 \\ \textbf{action } h1 \ :- \ \textit{true} \rightarrow \textit{skip} \\ \textbf{action } h2 \ :- \ \textit{true} \rightarrow x := 1 \\ \textbf{action } l \ :- \ x = 0 \rightarrow \textit{skip} \end{array} \right)$$

With $H = \{h1, h2\}$ and $M = \{h2\}$ we need to check $(\mathcal{A}14 \underset{H}{\|} FINITE_{H-M}) \setminus H$. We have:

$$\left(\begin{array}{l} \textbf{var } x : \{0,1\};\ \textit{count} : \mathbf{N} \\ \textbf{initially } x := 0;\ \textit{count} :\in \mathbf{N} \\ \textbf{action } l \ :- \ x = 0 \rightarrow \textit{skip} \\ \textbf{internal } h1 \ :- \ \textit{count} > 0 \rightarrow \textit{count} := \textit{count} - 1 \\ \textbf{internal } h2 \ :- \ \textit{false} \end{array} \right)$$

Hiding H produces a non-divergent, deterministic system, so $\mathcal{A}14$ is secure for actions other than $h2$. ♮

5.4 Security policies

The approach of the previous section begins to address the question of an independent security policy. For any system $\mathcal{A}$, the appropriate high-level user action is defined and then the combined system is checked for nondeterminism. However, the security policy is not straightforwardly reflected in the high-level user behaviour (it may not be immediately apparent, for example, that $FINITE_H$ encapsulates lazy deterministic security). Also, it is

only appropriate for conditional security where the condition can be defined as a structural requirement on the behaviour of H. For instance, suppose we wish to specify a system in which high- and low- level users may both send and receive messages. Within a certain time of the generation of each message no information flow from high to low may occur. However, after a period of time the messages from high-level users are no longer considered sensitive and there is no further need to protect them. In this situation it is not the actions of the high-level user alone which determine the condition, but some aspect of the system state will also be involved.

5.4.1 Other ways of restricting the system

In the following examples we investigate some of the ways in which the security requirements of a system can be represented in an action system.

Example 42 This is a simple aggregation problem in which a structural restriction is placed on L. The system is regarded as insecure if the low-level user makes more than ten interactions with the system. This could represent, for example, the low-level user gaining a number of separate bits of information which together might be considered a security threat. This restriction can be made for action system $\mathcal{A}$ by:

$$\mathcal{A} \|_L AGGR_L$$

where

$$AGGR_L \mathrel{\widehat{=}}$$

$$\left(\begin{array}{l} \textbf{var } ints : \mathbb{N} \\ \textbf{const } limit = 10 \\ \textbf{initially } ints := 0 \\ \textbf{action } l_i \in L \ :- \ ints < limit \ \rightarrow ints := ints + 1 \end{array}\right)$$

The limit can be set to the required value for any particular case. ♮

Example 43 In this example the agreement of the system manager is required for all operations from a set, S. Again for action system $\mathcal{A}$ we have:

$$\mathcal{A} \|_S AGREE_S$$

where

$$AGREE_S \mathrel{\widehat{=}}$$

$$\left(\begin{array}{l} \textbf{var } \{flag_{s_i} : \{yes, no\} \mid s_i \in S\} \\ \textbf{initially } \forall\, s_i \bullet flag_{s_i} := no \\ \textbf{action } s_i \in S \ :- \ flag_{s_i} = yes \rightarrow flag_{s_i} := no \\ \textbf{action } allow_{s_i} \ :- \ true \rightarrow flag_{s_i} := yes \end{array} \right)$$

Example 44 If explicit restrictions on the state are needed, an invariant can be used. This becomes an implicit part of the guard and must be reestablished by the execution of any action. A convenient way to record an invariant on the state is by using a Z schema. This example gives an invariant schema for a simple Chinese Wall policy. Various consultants, represented by the given set *CONSULTANT*, are employed by a number of different organisations from the set *ORGANISATION*. There may be conflicts of interest between the different organisations. Security is maintained by ensuring that a consultant employed by one organisation does not simultaneously carry out work for any competing organisation. The state consists of a set of conflict of interest classes and a function showing which organisations each consultant is currently working for.

$[CONSULTANT, ORGANISATION]$

CWALL

$$\begin{array}{l} coiclass : \mathbb{P}\, \mathbb{P}\, ORGANISATION \\ workson : CONSULTANT \rightarrow \mathbb{P}\, ORGANISATION \\ \hline \forall\, c : CONSULTANT;\ o1, o2 : ORGANISATION \bullet \\ \quad \{o1, o2\} \subseteq workson\ c \Rightarrow \\ \quad\quad \neg\, (\exists\, cl : coiclass \bullet \{o1, o2\} \subseteq cl) \end{array}$$

These approaches can all be combined with the determinism checks for non-interference.

5.4.2 Relaxing the conditions

The deterministic conditions debar any internal choices from influencing the actions of L. For instance, with:

$$\mathcal{A}15 \mathrel{\widehat{=}} \left(\begin{array}{l} \textbf{var } x : \{0,1\} \\ \textbf{initially } x :\in \{0,1\} \\ \textbf{action } l1 \ :- \ x = 0 \rightarrow \textit{skip} \\ \textbf{action } l2 \ :- \ x = 1 \rightarrow \textit{skip} \\ \vdots \end{array} \right)$$

$\mathcal{A}15$ is insecure since it is nondeterministic towards L. The justification is that this system could be refined by one in which the nondeterminism is resolved by interference from the high-level actions. The internal nondeterminism could also arise through internal actions. In either case, the possibility of unfortunate refinements means that the system is labelled as insecure. It may sometimes be useful to use a different, more liberal condition. The current ones say that there must be no nondeterminism towards L which *could* be influenced by H. A more generous condition might check that there is no such nondeterminism which *is* influenced by H. This is a retrograde step in terms of the refinement paradox but it is necessary in order to avoid, for example, explaining the whole of a key-generating mechanism at the top level. It does certainly put an onus on the user to re-prove the security property at a lower level to ensure that no insecurity is introduced. The case studies of Chapters 7 and 8 return to this topic.

5.5 Bi-directional channels

One aspect of action systems syntax not mentioned so far is the use of bi-directional channels. Butler [20] provides for these by extending the set *dir*

to:

$$dir \mathrel{\widehat{=}} \{in, out, inout\}$$

If a is a bi-directional channel then $a.(j,k)$ represents the action a with input value j and output value k. A particular action can be calculated for a given bi-directional channel in the following way:

Definition 31 *(Butler) If* $dir(a) = inout$ *then:*

$$\mathcal{A}_{a.(j,k)} \mathrel{\widehat{=}} (\textbf{local } x, y \bullet [(x,y) = (j,k)]\ \mathcal{A}_a\ [(x,y) = (j,k)])$$

The existing definitions can be used with bi-directional channels if *commgd* is extended in the following way:

Definition 32 *(Butler) If* $dir(a) = inout$ *then for* $S \subseteq \mathcal{W}$*:*

$$commgd(\mathcal{A}_{a.S}) \mathrel{\widehat{=}} (\exists\, x \in fst(S) \bullet gd(\mathcal{A}_a) \wedge wp(\mathcal{A}_a, y \in S_x))$$

where

$$fst(S) \mathrel{\widehat{=}} \{j \mid (\exists\, k \bullet (j,k) \in S)\}$$
$$S_J \mathrel{\widehat{=}} \{k \mid (j,k) \in S\}$$

Example 45

$$\mathcal{A}16 \mathrel{\widehat{=}}$$
$$\left(\begin{array}{l} \textbf{var} \\ \textbf{initially } skip \\ \textbf{action } a \textbf{ in } i? : \{0,1\} \textbf{ out } o! : \{0,1\}\ :- true \rightarrow o! := flip\ i? \end{array}\right)$$

where $flip\ 0 = 1$ and $flip\ 1 = 0$. The action $\mathcal{A}16_{a.(0,1)}$ with input 0 and output 1 is:

$$(\textbf{local } i?, o! \bullet [(i?, o!) = (0,1)]\ \mathcal{A}16_a\ [(i?, o!) = (0,1)])$$
$$\equiv (\textbf{local } i?, o! \bullet [(i?, o!) = (0,1)]\ i? = 0 \rightarrow o! := flip\ i?)$$
$$\equiv (\textbf{local } i?, o! \bullet i? = 0 \wedge o! = 1 \rightarrow o! := flip\ i?)$$
$$\equiv true \rightarrow skip$$

On the other hand, since $\mathcal{A}16_a$ can never satisfy a postcondition which requires $i?$ and $o!$ both to have the same value, we have that $\mathcal{A}16_{a.(0,0)}$ is a miracle. ♮

If bi-directional channels are used there is no need for the mixed security property of Section 5.3. Signal events are represented as the output part of an **inout** channel, so no delay in the system will be observable.

5.5.1 The secure multiple stack system

This example is adapted from [87] where a trace specification is used to define multi-level stacks. Here, an action system is used to specify a system in which each classification has an associated stack. The set *CLASS*, of classifications, is assumed to be linearly ordered. There is a set *USER* of users, each of which has an associated classification specifying their clearance level. We will assume that these two sets are finite. User classification is represented by the function:

$$clear : USER \rightarrow CLASS$$

Each stack is modelled as a sequence of *DATA*. The state of the system is noted in the **var** part of an action system. Here we use a Z schema to define the state, using the schema name within the action system. This is a convenient way to make the action system more readable, particularly when the state is more complex with a lengthy invariant. Note that this is just one possible way to denote the state: it is not an integral part of the action systems approach. Any notation incorporated in this way will be used in a strictly limited way within the framework of an action system. The state is defined by the following schema:

S

$$stack : CLASS \rightarrow \mathrm{seq}\, DATA$$

We also define the possible outputs from the system as either an item of data or one of two error messages:

$$REP ::= Rep\langle\langle DATA\rangle\rangle \mid empty \mid error$$

$MultiStack \mathrel{\widehat{=}}$

$$
\left(\begin{array}{l}
\textbf{var } S \\
\textbf{initially } stack : [\mathrm{ran}\ stack = \{\langle\rangle\}] \\
\textbf{action } push \textbf{ in } u? : USER;\ sc? : CLASS;\ d? : DATA\ :- \\
\quad (clear\ u?) \leq sc? \rightarrow (stack\ sc?) := (stack\ sc?)^{\frown}\langle d?\rangle \\
\textbf{action } pop \textbf{ in } u? : USER;\ sc? : CLASS\ :- \\
\quad (clear\ u?) \leq sc? \rightarrow \ \mathit{if}\ (stack\ sc?) \neq \langle\rangle \\
\qquad\qquad\qquad\qquad \mathit{then}\ (stack\ sc?) := \mathit{front}\ (stack\ sc?) \\
\qquad\qquad\qquad\qquad \mathit{else\ skip} \\
\textbf{action } top \textbf{ in } u? : USER;\ sc? : CLASS \textbf{ out } r! : REP\ :- \\
\quad true \rightarrow \ \mathit{if}\ (clear\ u?) < sc? \\
\qquad\qquad \mathit{then}\ r! := error \\
\qquad\qquad \mathit{else}\ \mathit{if}\ (stack\ sc?) = \langle\rangle \\
\qquad\qquad\qquad \mathit{then}\ r! := empty \\
\qquad\qquad\qquad \mathit{else}\ r! := Rep(\mathit{last}\ (stack\ sc?))
\end{array}\right)
$$

Figure 5.1 Multi-level stacks

There are three operations on stacks:

- *PUSH* allows a user to put a given data item on the top of a specified stack
- *TOP* returns the top data item from the specified stack to the requesting user but does not alter the stack
- *POP* removes the top item of data from the specified stack

The system must not allow users to read above their clearance or to write below it. The action system *MultiStack* of Figure 5.1 specifies the system.

The *MultiStack* action *top* is an example of the use of bidirectional channels, introduced in the previous section. The actions for any classification, cl, are those with $clear\ u? = cl$. Let H_{cl} be the set of all actions with $clear\ u? \geq cl$. To satisfy the eager security condition we would need:

$$(MultiStack \setminus H_{cl})\ deterministic \qquad \text{for each } H_{cl}$$

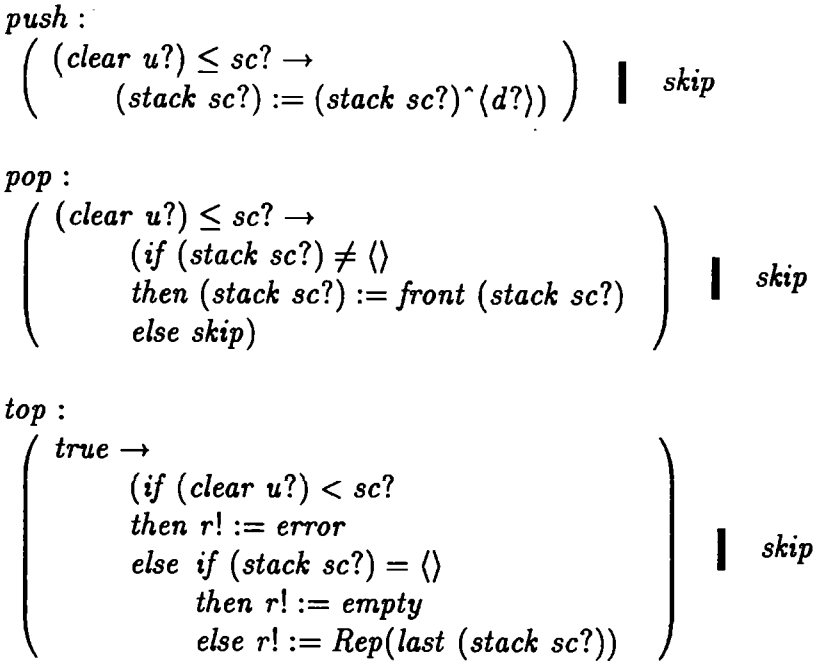

Figure 5.2 Actions of $obs_{H_{cl}}(MultiStack)$ for *clear* $u? \geq cl$

Hiding the actions in any H_{cl} causes divergence. For example, the H_{cl} action *top* for user of clearance *cl* is always enabled. Hence the specification is not eagerly secure.

Next we check for lazy deterministic security by considering the system $obs_{H_{cl}}(MultiStack)$. For any arbitrary classification *cl*, the high-level actions H_{cl}, that is those for *clear* $u? \geq cl$, will be as shown in Figure 5.2. Lower level actions remain unchanged.

Actions for *clear* $u? \geq cl$ are always enabled so the system can only be nondeterministic if there is some lower level action whose guard could, through internal choice, either be enabled or not after a particular trace. Hence we consider the actions for lower level users, that is, with *clear* $u? < cl$.

The guard for all lower level *push* and *pop* actions is *clear* $u? \leq sc?$. The clearance is never changed, so this is not something that could be enabled or otherwise by any trace of actions. For *top* the situation is a little more complicated. The guard is *true* but the definition of a channel requires that the guard has to be calculated for specific values of inputs $u?$ and $sc?$ and

output $r!$.

Suppose the input user is i and the input class is c. Since we are dealing with a lower-level *top* action we are assuming that $clear\ i < cl$. There are three possible cases for the output reply. Firstly, consider the case of an "error" output. Using Definition 31 the specific *top* action in this case is:

$$
\begin{array}{l}
(\textbf{local}\ u?, sc?, r! \bullet \\
\quad [(u?, sc?, r!) = (i, c, error)] \\
\quad obs_{H_{cl}}(MultiStack)_{top} \\
\quad [(u?, sc?, r!) = (i, c, error)] \\
)
\end{array}
$$

$$
\begin{array}{l}
\equiv (\textbf{local}\ u?, sc?, r! \bullet \\
\quad [(u?, sc?, r!) = (i, c, error)] \\
\quad true \rightarrow\ if\ (clear\ u?) < sc? \\
\qquad\qquad then\ r! := error \\
\qquad\qquad else\ if\ (stack\ sc?) = \langle\rangle \\
\qquad\qquad\quad then\ r! := empty \\
\qquad\qquad\quad else\ r! := Rep(last\ (stack\ sc?)) \\
\quad [(u?, sc?, r!) = (i, c, error)] \\
)
\end{array}
$$

$$
\begin{array}{l}
\equiv (\textbf{local}\ u?, sc?, r! \bullet \\
\quad [(u?, sc?, r!) = (i, c, error)] \\
\quad u? = i \wedge sc? = c \wedge (clear\ u?) < sc? \rightarrow r! := error \\
)
\end{array}
$$

$$
\equiv (clear\ i) < c \rightarrow skip
$$

So the guard of *top* for these values is $clear\ i < c$. Again, this is fully determined by the actual inputs and cannot be enabled or disabled by any preceding trace of actions.

In the second case, $r! = empty$. The action this time is:

$$
\begin{aligned}
&(\textbf{local } u?, sc?, r! \bullet \\
&\qquad [(u?, sc?, r!) = (i, c, empty)] \\
&\qquad obs_{H_{cl}}(MultiStack)_{top} \\
&\qquad [(u?, sc?, r!) = (i, c, empty)] \\
&) \\
&\\
&\equiv (\textbf{local } u?, sc?, r! \bullet \\
&\qquad [(u?, sc?, r!) = (i, c, empty)] \\
&\qquad u? = i \wedge sc? = c \wedge (clear\ u?) \geq sc? \wedge (stack\ sc?) = \langle\rangle \\
&\qquad\qquad \rightarrow r! := empty \\
&\quad) \\
&\\
&\equiv (clear\ i) \geq c \wedge (stack\ c) = \langle\rangle \rightarrow skip
\end{aligned}
$$

The first conjunct of this is, as before, determined by the inputs alone. If it is false, the action is not enabled. So assume it is true, that is: $(clear\ i) \geq c$. Since the action is a lower level one we know that $cl > clear\ i$. Combining this with the previous assumption gives $cl > c$, which means that the classification of the requested stack is strictly less than the high level classification, cl. Any action at a level $\geq cl$ cannot alter the stack at level c since the high-level actions can only affect stacks whose classification is as least as great as their own. This means that no high-level action has any influence over whether $(stack\ c) = \langle\rangle$ or not. So the internal actions made by the high-level actions of $obs_{H_{cl}}(MultiStack)$ do not affect the enabling of the low-level guards in this case.

The guard for the third case is determined as follows:

$$
\begin{aligned}
&(\textbf{local } u?, sc?, r! \bullet \\
&\quad [(u?, sc?, r!) = (i, c, Rep(d))] \\
&\quad obs_{H_{cl}}(MultiStack)_{top} \\
&\quad [(u?, sc?, r!) = (i, c, Rep(d))] \\
&) \\
&\equiv (\textbf{local } u?, sc?, r! \bullet \\
&\quad [(u?, sc?, r!) = (i, c, Rep(d))] \\
&\quad u? = i \wedge sc? = c \wedge (clear\ u?) \geq sc? \\
&\qquad \wedge (stack\ sc?) \neq \langle\rangle \wedge last(stack\ sc?) = d \\
&\qquad\quad \rightarrow r! := Rep(d) \\
&) \\
&\equiv (clear\ i) \geq c \wedge last(stack\ c) = d \rightarrow skip
\end{aligned}
$$

As in the previous case, no high-level action can influence the enabling of the guard of this action.

This has shown that the internal choices of high-level actions can have no effect on the enablement of low-level actions. There is no other way that nondeterminism could arise in $obs_{H_{cl}}(MultiStack)$ (no internal actions, no low-level nondeterminism) and hence the system is deterministic. Thus *MultiStack* meets the condition for lazy deterministic security.

5.5.2 Representation of bidirectional channels

Representing bidirectional channels[1] in the manner described above has the advantage of removing the need for the mixed deterministic security property for dealing with high-level output actions. However, such a representation may be problematic from the point of view of the development of a system. Given this interpretation, a bidirectional channel must always be refined as a unit and cannot be decomposed in a way which splits up input and

[1]Thanks to Michael Butler for discussion on this point.

output. This makes the possibilities for refinement greatly limited. Output values must be regarded as being calculated simultaneously with receiving the input. There is no scope for receiving input and then following on with internal processing which will determine an output. If this is regarded as a problem it would be possible to provide a model for bidirectional channels which interprets them semantically as two separate actions: input followed by output. The situation for the security definitions would then be the same as for the original CSP ones and mixed security conditions would be required.

In practical terms, the restrictions on refinement mean that bidirectional channels are perhaps not as useful as they might first appear. Although they work well for the examples given here, they do not appear in the case studies of later chapters because of the inflexibility caused by their introduction. Hence, whilst bidirectional chanels provide a neat characterisation at the specification level, the practicalities of system development mean that they are not such a good choice for general use.

5.6 Summary

This chapter introduced the definitions necessary for using deterministic security properties for action systems. The two main properties, for eager and lazy forms of the definition, were given. These were shown to correspond to the equivalent properties in CSP. Other forms of security (including strong security, mixed security and conditional security) were also addressed. Corresponding to the alternative representation of deterministic security properties in CSP, Section 5.3 provided a characterisation in terms of the allowed high-level user activity. Examples throughout the chapter illustrated these concepts as they were introduced and showed how the security definitions are applied.

The provision of security definitions is necessary for the assessment of security at any given level. Another important factor for system development is the relationship of a secure system at one level to a refinement of that system. This is explored in the next chapter.

Chapter 6

Refining secure action systems

This chapter presents refinement rules for action systems. The basis for action system refinement corresponding to CSP failures-divergences refinement was presented by Woodcock and Morgan [132]. Given a simulation relation between abstract and concrete states obeying certain properties, then soundness with respect to failures-divergences is guaranteed. This work has been taken further by Butler [20, 18] who defines refinement for action systems with internal actions and for value-passing action systems.

This chapter gives the definitions with examples of their use and considers how refinement is related to the deterministic security properties. One important aspect of action system refinement is the technique of parallel decomposition and that is also covered here. Given the basic definitions for action system refinement it is possible to deduce a number of simplified rules which can be applied in particular circumstances. A number of these which prove useful for later chapters are introduced here.

6.1 Refinement and simulation

Refinement is the process of moving from an abstract representation of a system to a more concrete one. In CSP terms, process Q is a refinement of

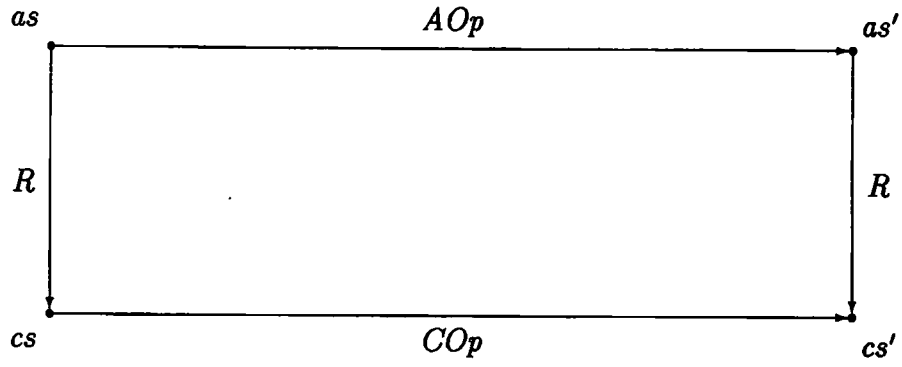

Figure 6.1 Abstract and concrete states related by simulation R

process P if it displays a subset of the possible behaviours of P, that is:

$$fails(Q) \subseteq fails(P)$$
$$divs(Q) \subseteq divs(P)$$
$$infs(Q) \subseteq infs(P)$$

A deterministic system has no proper refinements.

An action system considers state as well as events. In this case, part of the process of refinement involves moving from an abstract representation of the state of the system to a more concrete one. This is referred to as data refinement. Each concrete operation must correspond to a distinct abstract operation in a way which ensures that the behaviours of the concrete system form a subset of the behaviours of the abstract system.

Refinement can be characterised by means of simulation. A simulation is a relation between abstract and concrete states which must obey certain properties (detailed below). One system is refined by another if and only if a simulation exists between them.

In general terms, a forwards simulation between abstract state AS and concrete state CS is a relation $R : AS \leftrightarrow CS$ for which the effect of applying any concrete operation, COp, must be matched by application of the equivalent abstract operation, AOp, to a related state. This is often depicted as in Diagram 6.1 in which dashes represent the state after the operation.

Starting from as, if R followed by COp can result in state cs', then

it must also be possible to arrive at cs' by AOp followed by R. Hence, $R ; COp \subseteq AOp ; R$. This is known as forwards simulation (since any forward move of the concrete system must be matched by a corresponding move in the related abstract state) or as downwards simulation (given the direction of the arrows, down from abstract to concrete).

A complementary relationship is provided by backwards (or upwards) simulation. Again, with reference to Figure 6.1, starting from concrete state cs, $COp ; R^{-1}$ must be matchable by $R^{-1} ; AOp$. This can be viewed as moving upwards from the concrete to the abstract. Given any concrete state cs in the range of R, if COp can arrive at cs' then a matching move of AOp can reach a related abstract state. That is: $COp ; R^{-1} \subseteq R^{-1} ; AOp$. This interpretation of looking back to see how a given concrete state was reached leads to the term backwards simulation.

A good general introduction to refinement and to simulation is provided by Woodcock and Davies [133].

6.2 Refinement for basic action systems

In [132] Morgan and Woodcock give forwards and backwards simulation rules for basic action systems which are shown to be sound and jointly complete with respect to CSP refinement[1]. However, by using a general representation function, Butler [20] accommodates both types of simulation with a single definition. Forwards and backwards simulation can be distinguished by the nature of the representation function. This single, combined approach allows a neater, uniform presentation of the theory for the general case. However, in practice forwards simulation is often sufficient to prove most refinements. By narrowing the scope to consider only this method of refinement, developments can be greatly assisted by the use of refinement calculus techniques. Also, refinement rules can be simplified when the simulation relation is functional from concrete to abstract states.

[1] Although a completeness theorem is proved for the basic action systems of [132], this result has not been extended to value-passing action systems.

Here, definitions are first given for both forms of simulation with examples to show the difference between them. Following this, throughout the remainder of the chapter the rules for forward simulation will be used.

6.2.1 Forwards simulation for basic action systems

Action b refines action a if the possible behaviours of b are a subset of the possible behaviours of a. Nondeterminism may be eliminated, but no 'extra' activity may be introduced. For data refinement, the change of representation of state from abstract to concrete must also be taken into account. Suppose abstract state S and concrete state T are related by R (known as the refinement relation, retrieve relation or abstraction invariant). If b refines a with respect to R we write $a \preceq_R b$. In the case that R is a forwards simulation this is defined as follows:

Definition 33 *For any postcondition α with no free occurrences of concrete variables, $a \preceq_R b$ if:*

$$(\exists S \bullet R \wedge wp(a, \alpha)) \Rrightarrow wp(b, (\exists S \bullet R \wedge \alpha))$$

Here, $\Rrightarrow$ represents entailment (that is, implication true in all states).

The corresponding condition for forwards simulation between initialisations is:

$$wp(\mathcal{A}_i, \alpha) \Rrightarrow wp(\mathcal{B}_i, (\exists S \bullet R \wedge \alpha))$$

In this case we write: $\mathcal{A}_i \preceq'_R \mathcal{B}_i$.

For action system $\mathcal{B}$ to be a refinement of action system $\mathcal{A}$, both must share the same alphabet, with $\mathcal{A}_a$ refined by $\mathcal{B}_a$ for each a in the joint alphabet. The initialisation of $\mathcal{A}$ must also be refined by that of $\mathcal{B}$. A third requirement concerns the relationship between guards. The guard of the concrete action must be no stronger than that of the abstract action, otherwise additional refusals could be introduced. This motivates the following definition of action system refinement.

Definition 34 *Forwards simulation(Butler) Action system* $\mathcal{A} = (A, S, \mathcal{A}_i, \mathcal{A}_A)$ *is refined by action system* $\mathcal{B} = (A, T, \mathcal{B}_i, \mathcal{B}_A)$ *if there is a relation* R *between them such that:*

1. $\mathcal{A}_i \preceq'_R \mathcal{B}_i$
2. $\mathcal{A}_a \preceq_R \mathcal{B}_a$ *for each* a *in* A
3. $(\exists S \bullet R \wedge gd(\mathcal{A}_a)) \Rrightarrow gd(\mathcal{B}_a)$ *for each* a *in* A

If these conditions are satisfied we write $\mathcal{A} \sqsubseteq_R \mathcal{B}$.

6.2.2 Examples

Example 46 Take $\mathcal{A}3$ as defined above:

$$\mathcal{A}3 \mathrel{\widehat{=}} \left(\begin{array}{l} \textbf{var } x : \{0,1\} \\ \textbf{initially } x := 0 \\ \textbf{action } h \ :- \ x = 0 \rightarrow x :\in \{0,1\} \\ \textbf{action } l \ :- \ x = 1 \rightarrow x := 0 \end{array} \right)$$

A purely algorithmic refinement of this (that is, with no change of state) is:

$$\mathcal{B}1 \mathrel{\widehat{=}} \left(\begin{array}{l} \textbf{var } x : \{0,1\} \\ \textbf{initially } x := 0 \\ \textbf{action } h \ :- \ x = 0 \rightarrow x := 1 \\ \textbf{action } l \ :- \ x = 1 \rightarrow x := 0 \end{array} \right)$$

This has simply limited the nondeterminism by making a definite choice for the value of x in action h. Therefore $\mathcal{A}3 \sqsubseteq \mathcal{B}1$. Resolving the nondeterminism by setting x to 0 in h would also produce an acceptable refinement. In this case, only h actions would ever be enabled.

Any introduction of nondeterminism will cause condition 2 of Definition 34 to be false. For instance, $\mathcal{B}1$ is not refined by $\mathcal{A}3$. This is because:

$$wp(\mathcal{B}1_h, \alpha) \equiv x = 0 \Rightarrow \alpha[1/x]$$

and:

$$wp(\mathcal{A}3_h, \alpha) \equiv x = 0 \Rightarrow (\alpha[0/x] \vee \alpha[1/x])$$

and so:

$$wp(\mathcal{B}1_k, \alpha) \not\Rightarrow wp(\mathcal{A}3_k, \alpha)$$

$\natural$

Example 47 Here data refinement is used as well. With:

$$\mathcal{A}17 \mathrel{\widehat{=}} \left(\begin{array}{l} \textbf{var } s : \mathbb{P}\ \mathbb{N};\ n : \mathbb{N} \\ \textbf{initially } s := \emptyset;\ n :\in \mathbb{N} \\ \textbf{action } a1\ :- \mathit{true} \rightarrow n :\in \mathbb{N} - s;\ s := s \cup \{n\} \\ \textbf{action } a2\ :- s \neq \emptyset \rightarrow n :\in s;\ s := s - \{n\} \end{array} \right)$$

$$\mathcal{B}2 \mathrel{\widehat{=}} \left(\begin{array}{l} \textbf{var } m : \mathbb{N} \\ \textbf{initially } m := 0 \\ \textbf{action } a1\ :- \mathit{true} \rightarrow m := m + 1 \\ \textbf{action } a2\ :- m \neq 0 \rightarrow m := m - 1 \end{array} \right)$$

The concrete variable m is simply a number which may be incremented or decremented. The refinement relation is:

$$R \mathrel{\widehat{=}} s = (1 \mathinner{..} m) \wedge n = m$$

The refinement of $\mathcal{A}17$ by $\mathcal{B}2$ is established by checking the three conditions from Definition 34. Proof of this is given in Appendix D. Thus:

$$\mathcal{A}17 \sqsubseteq_R \mathcal{B}2$$

$\natural$

Example 48 In this example the abstract system describes a very basic keyserver. We specify a single user setting up a session and obtaining a key. The key allocated by the keyserver is drawn from the following given set which is assumed to be nonempty:

$$[KEY]$$

The data type STATUS is defined:

$$STATUS ::= no \mid yes \mid send$$

with the following intended meanings:

no no session is established with the keyserver.

yes a session is established with the keyserver (but no key has yet been sent).

send a session is established and a key has been sent.

The abstract system is defined:

$$KeyServer1 \mathrel{\widehat{=}}$$

$$\left(\begin{array}{l} \textbf{var } k1 : KEY;\ s1 : STATUS \\ \textbf{initially } s1 := no;\ k1 :\in KEY \\ \textbf{action } startsession\ :- s1 = no \rightarrow s1 := yes \\ \textbf{action } sendkey\ :- s1 = yes \rightarrow k1 :\in KEY;\ s1 := send \end{array}\right)$$

When the user requests the start of a session the status is set to *yes*. When the user requests that the key be sent, the keyserver chooses a key and the status is set to *send*. (This abstracts away from the details of actually sending the key).

The action system *KeyServer2* describes a similar system, the difference being that *KeyServer2* decides which key is to be allocated as soon as the session is established. The key is not communicated until it is requested, so from the user's point of view, the two systems are equivalent. The status variables $s1$ and $s2$ ensure that the same succession of actions within the system is maintained.

$$KeyServer2 \mathrel{\widehat{=}}$$

$$\left(\begin{array}{l} \textbf{var } k2 : KEY;\ s2 : STATUS \\ \textbf{initially } s2 := no;\ k2 :\in KEY \\ \textbf{action } startsession\ :- s2 = no \rightarrow s2 := yes;\ k2 :\in KEY \\ \textbf{action } sendkey\ :- s2 = yes \rightarrow s2 := send \end{array}\right)$$

For the relation between the two systems, at the point at which the key is communicated it must be possible for both systems to supply the same key. This leads to the definition:

$$R \mathrel{\widehat{=}} (s1 = s2) \wedge (s1 = s2 = send \Rightarrow k1 = k2)$$

All the refinement conditions are satisfied, as confirmed in Appendix D, and so:

$$KeyServer1 \sqsubseteq_R KeyServer2$$

$\natural$

Since the two versions of the key server are so similar it may be useful to investigate whether refinement holds in the other direction: that is, does $KeyServer2 \sqsubseteq KeyServer1$? However, a problem arises when checking Condition 2 for *sendkey* where:

$$\begin{aligned}
LHS &\equiv (\exists\, k2 : KEY;\ s2 : STATUS \bullet R \wedge \\
&\qquad wp(s2 = yes \rightarrow s2 := send, \alpha)) \\
&\equiv (\exists\, k2 : KEY;\ s2 : STATUS \bullet (s1 = s2) \wedge \\
&\qquad (s1 = s2 = send \Rightarrow k1 = k2) \wedge \\
&\qquad (s2 = yes \Rightarrow \alpha[send/s2])) \\
&\equiv (\exists\, k2 : KEY \bullet (s1 = send \Rightarrow k1 = k2) \wedge (s1 = yes \Rightarrow \alpha[send/s2]))
\end{aligned}$$

$$\begin{aligned}
RHS &\equiv wp(s1 = yes \rightarrow k1 :\in KEY;\ s1 := send, \\
&\qquad (\exists\, k2 : KEY;\ s2 : STATUS \bullet R \wedge \alpha)) \\
&\equiv s1 = yes \Rightarrow (\forall\, k1 : KEY \bullet (\exists\, k2 : KEY;\ s2 : STATUS \bullet \\
&\qquad s2 = send \wedge k1 = k2 \wedge \alpha)) \\
&\equiv s1 = yes \Rightarrow (\forall\, k1 : KEY \bullet \alpha[k1, send/k2, s2]) \\
&\equiv s1 = yes \Rightarrow (\forall\, k2 : KEY \bullet \alpha[send/s2])
\end{aligned}$$

Therefore $LHS \not\equiv RHS$. The LHS does imply that:

$$\begin{aligned}
&(\exists\, k2 : KEY \bullet s1 = yes \Rightarrow \alpha[send/s2]) \wedge \\
&s1 = yes \Rightarrow (\exists\, k2 : KEY \bullet \alpha[send/s2])
\end{aligned}$$

but this is not strong enough to imply the RHS which demands that the result hold for all $k2$. The problem in this case is that $KeyServer1$ delays the choice

of key. Any choice made by *KeyServer*1 could previously have been made by *KeyServer*2. Hence there is an abstract state from which the abstract *sendkey* could have ended up with the same result. However, this is not good enough for forwards simulation, and the conditions cannot be satisfied. Since *KeyServer*1 and *KeyServer*2 have exactly the same failures, divergences and infinite traces in their CSP interpretations, this illustrates the fact that forwards simulation is not complete with respect to CSP refinement. The complementary relation of backwards simulation is needed (see Example 49). Although forwards simulation proves sufficient for many applications it is worth noting that in cases such as this where choice is delayed, backwards simulation will be needed.

6.2.3 Backwards simulation for basic action systems

For backwards simulation a different interpretation is given to the refinement of actions. For actions a and b and abstraction invariant R we will write $a \overleftarrow{\preceq}_R b$ if:

$$(\forall S \bullet R \Rightarrow wp(a, \alpha)) \equiv\!\!\!\Rightarrow wp(b, (\forall S \bullet R \Rightarrow \alpha))$$

for all postconditions, α, with no free ocurrences of concrete variables. Similarly, a backwards simulation between initialisations $\mathcal{A}_i$ and $\mathcal{B}_i$ is written $\mathcal{A}_i \overleftarrow{\preceq}'_R \mathcal{B}_i$ and defined as:

$$wp(\mathcal{A}_i, \alpha) \equiv\!\!\!\Rightarrow wp(\mathcal{B}_i, (\forall S \bullet R \Rightarrow \alpha))$$

where again, α may be any post condition with no free occurrences of concrete variables. The definition of backwards simulation makes use of this:

Definition 35 *Backwards simulation (Butler) Action system $\mathcal{A} = (A, S, \mathcal{A}_i, \mathcal{A}_A)$ is refined by action system $\mathcal{B} = (A, T, \mathcal{B}_i, \mathcal{B}_A)$ if there is a relation R between them such that:*

1. $\mathcal{A}_i \overleftarrow{\preceq}'_R \mathcal{B}_i$
2. $\mathcal{A}_a \overleftarrow{\preceq}_R \mathcal{B}_a$ *for each a in A*
3. $(\forall S \bullet R \Rightarrow gd(\mathcal{A}_s)) \equiv\!\!\!\Rightarrow gd(\mathcal{B}_s)$ *for each $s \subseteq A$*

If these conditions are satisfied we write $\mathcal{A} \sqsubseteq_R \mathcal{B}$.

Example 49 With *KeyServer*1 and *KeyServer*2 as defined in the previous example:

$$KeyServer2 \sqsubseteq_R KeyServer1$$

under backwards simulation. A proof is given in Appendix D. Given the refinement in both directions the two systems *KeyServer*1 and *KeyServer*2 are essentially equivalent. This can also be seen by consideration of their failures-divergences interpretations. We return to the *KeyServer* example later in the chapter showing how a simple system can be defined and refined to output the chosen key. ♮

6.3 Refinement and internal actions

For action systems with internal actions the refinement rule needs to address the effect of hidden actions. This is achieved by modifying the conditions with the $*$ operator defined in Chapter 4.

Definition 36 *(Butler) Action system* $\mathcal{A} = (A, S, \mathcal{A}_i, \mathcal{A}_A, \mathcal{A}_I)$ *is refined by action system* $\mathcal{B} = (A, T, \mathcal{B}_i, \mathcal{B}_a, \mathcal{B}_J)$ *if there is a relation* R *between them such that:*

1. $\mathcal{A}_i * I \preceq'_R \mathcal{B}_i * J$
2. $\mathcal{A}_a * I \preceq_R \mathcal{B}_a * J$ *for each* a *in* A
3. $(\exists S \bullet R \wedge gd_I(\mathcal{A}_a)) \Rrightarrow gd_J(\mathcal{B}_a)$ *for each* a *in* A

Example 50 The *Countdown* specification with automatic resetting was defined in Example 20 as:

$$\left(\begin{array}{l} \textbf{var } daysleft : \mathbb{N} \\ \textbf{initially } daysleft := 100 \\ \textbf{action } newday \ :- \ daysleft > 0 \rightarrow daysleft := daysleft - 1 \\ \textbf{internal } reset \ :- \ daysleft = 0 \rightarrow daysleft := 100 \end{array}\right)$$

We will refer to this as *ConcCount*. It may be viewed as a refinement of the abstract action system *AbsCount* which is always willing to participate in the *newday* action:

$$AbsCount \mathrel{\widehat{=}} \left(\begin{array}{l} \textbf{var} \\ \textbf{initially } skip \\ \textbf{action } newday \ :- \ true \rightarrow skip \end{array} \right)$$

The refinement:

$$AbsCount \sqsubseteq_R ConcCount$$

may be verified by proof of the conditions of Definition 36 with refinement relation R defined simply as *true*. This is confirmed in Appendix D. ♮

Proof of these conditions can be unwieldy for large systems and in certain circumstances simplification is possible (see Section 6.5).

6.4 Refinement for value-passing action systems

Refinement may be defined between two value-passing action systems with the same set of channels and with identical channel directions and values of communication. The pattern of the definition for forwards simulation is the same as for the previous refinement definitions, with three conditions ensuring correctness of initialisation, correctness of actions and applicability of actions respectively.

Definition 37 *(Butler) Action system* $\mathcal{A} = (A, S, \mathcal{A}_i, \mathcal{A}_A, \mathcal{A}_I, dir)$ *is refined by action system* $\mathcal{B} = (A, T, \mathcal{B}_i, \mathcal{B}_a, \mathcal{B}_J, dir)$ *if there is a relation* R *between them such that:*

1. $\mathcal{A}_i * I \preceq'_R \mathcal{B}_i * J$
2. $\mathcal{A}_{a.i} * I \preceq_R \mathcal{B}_{a.i} * J$ *for each a.i in A*
3. $(\exists S \bullet R \wedge commgd_I(\mathcal{A}_a)) \Rrightarrow commgd_J(\mathcal{B}_a)$ *for each a in A*

If the action systems have no internal actions and output is deterministic then Definition 37 is equivalent to Definition 34 with Conditions 2 and 3 holding for each channel-value pair, $a.v$.

Example 51 Here the secure multi-stack of Example 5.5.1 is refined. Suppose the relationship between stacks and their classification is to be implemented by parallel arrays. The concrete stack may be defined:

$$
\begin{array}{|l}
\hline
CS \\
\hline
cclass : arr_l\ CLASS \\
cstack : arr_l\ (\mathrm{seq}\ DATA) \\
\hline
\mathrm{ran}\ cclass = CLASS \\
\hline
\end{array}
$$

where $arr_l\ X$ is an l-length sequence of X, l being the number of classifications in the system. The abstraction relation R is defined:

$$R \mathrel{\widehat=} CS \wedge S \wedge \forall\, cl : CLASS \bullet stack\ cl = cstack\ (cclass^{-1}\ cl)$$

The concrete system is shown in Figure 6.2. It has no internal actions and output is deterministic. The conditions of Definition 34 are checked in Appendix D showing that:

$$MultiStack \sqsubseteq ConcStack$$

It was shown above that *MultiStack* is secure with respect to the condition for lazy deterministic security. No new nondeterminism can be introduced by refinement, and hence *ConcStack* is also secure. ♮

6.5 Special refinement conditions

The refinement conditions given above can be hard to verify in practice. For instance, Definition 37 requires correspondence of each individual communication $\mathcal{A}_{a.i}$ and $\mathcal{B}_{a.i}$. As soon as the refinement rules are applied to particular systems it becomes apparent that there are many simplified rules which can be derived to be applied in specific circumstances. A number of these were

$$
\begin{array}{l}
ConcStack \mathrel{\widehat{=}} \\
\left(\begin{array}{l}
\textbf{var } CS \\
\\
\textbf{initially } cclass, cstack : \left[\begin{array}{l} \text{ran } cclass = CLASS \wedge \\ \text{ran } cstack = \{\langle\rangle\} \end{array}\right] \\
\\
\textbf{action } push \textbf{ in } u? : USER;\ sc? : CLASS;\ d? : DATA\ :- \\
\quad (clear\ u?) \leq sc? \rightarrow \\
\qquad (cstack\ (cclass^{-1}\ sc?)) := (cstack\ (cclass^{-1}\ sc?))^{\frown}\langle d?\rangle \\
\\
\textbf{action } pop \textbf{ in } u? : USER;\ sc? : CLASS\ :- \\
\quad (clear\ u?) \leq sc? \rightarrow \\
\qquad \mathit{if}\ (cstack\ (cclass^{-1}\ sc?)) \neq \langle\rangle \\
\qquad \mathit{then}\ (cstack\ (cclass^{-1}\ sc?)) \\
\qquad\qquad := front\ (cstack\ (cclass^{-1}\ sc?)) \\
\qquad \mathit{else\ skip} \\
\\
\textbf{action } top \textbf{ in } u? : USER;\ sc? : CLASS \textbf{ out } r! : REP\ :- \\
\quad true \rightarrow\ \mathit{if}\ (clear\ u?) < sc? \\
\qquad\qquad \mathit{then}\ r! := error \\
\qquad\qquad \mathit{else\ if}\ (cstack\ (cclass^{-1}\ sc?)) = \langle\rangle \\
\qquad\qquad\quad \mathit{then}\ r! := empty \\
\qquad\qquad\quad \mathit{else}\ r! := Rep(last\ (cstack\ (cclass^{-1}\ sc?)))
\end{array}\right)
\end{array}
$$

Figure 6.2 Concrete stack specification

noted by Butler [20] during the development of case studies. Derived rules for action systems are also calculated by Sinclair and Woodcock [123]. It is extremely useful to have a library of such laws available as it can often reduce the verification work required.

The following two rules (proved in [20]) are used in the case studies of Chapters 7 and 8. The first can be used instead of Definition 37 to verify forwards simulation for value-passing action systems.

Property 3 *Action system* $\mathcal{A} = (A, S, \mathcal{A}_i, \mathcal{A}_A, \mathcal{A}_I, \mathcal{A}_H, dir)$ *is refined by action system* $\mathcal{B} = (A, T, \mathcal{B}_i, \mathcal{B}_a, \mathcal{B}_J, dir)$ *with refinement relation* R *if:*

1. $\mathcal{A}_i \preceq'_R \mathcal{B}_i$
2. $\mathcal{A}_a \preceq_R \mathcal{B}_a$ *for each* a *in* A

3. $IT_H \preceq_R IT_J$

4. $(\exists S \bullet R \wedge \overline{wp}(IT_H, gd(\mathcal{A}_a))) \Rrightarrow \overline{wp}(IT_J, gd(\mathcal{B}_a))$
for each input action a

5. $(\exists S \bullet R \wedge \overline{wp}(IT_H, gd(\mathcal{A}_a) \wedge wp(\mathcal{A}_a, \phi))) \Rrightarrow$
$\overline{wp}(IT_J, gd(\mathcal{B}_a) \wedge wp(\mathcal{B}_a, \phi))$
for each output action a and predicate ϕ on the output variable

The second property can be used if the abstract system has no hidden actions. Each hidden action of the concrete system is shown to refine *skip*. Non-divergence of hidden actions is ensured by means of a variant (E, belonging to well-founded set WF) which is decreased by each concrete hidden action.

Property 4 *Action system $\mathcal{A} = (A, S, \mathcal{A}_i, \mathcal{A}_A, dir)$ is refined by action system $\mathcal{B} = (A, T, \mathcal{B}_i, \mathcal{B}_a, \mathcal{B}_J, dir)$ if there is a relation R between them such that for some well-founded set WF and variant E:*

1. $\mathcal{A}_i \preceq'_R \mathcal{B}_i$

2. $\mathcal{A}_a \preceq_R \mathcal{B}_a$ *for each a in A*

3. $skip \preceq_R \mathcal{B}_j$ *for each j in J*

4. $(\exists S \bullet R) \Rrightarrow E \in WF$

5. $(\exists S \bullet R) \wedge E = e \Rrightarrow wp(\mathcal{B}_j, E < e)$ *for each j in J*

6. $(\exists S \bullet R \wedge gd(\mathcal{A}_a)) \Rrightarrow gd(\mathcal{B}_a) \vee (\exists j \in J \bullet gd(\mathcal{B}_j))$
for each a in A

Another useful property allows a single internal action to be split into several internal actions.

Property 5 (Internal Split Rule) *Suppose action system $\mathcal{A}$ has internal action a and action system $\mathcal{B}$ is the same action system as $\mathcal{A}$ but with action a replaced by two internal actions, b and c, such that for any postcondition α:*

$$wp(\mathcal{A}_a, \alpha) \equiv wp(\mathcal{B}_b \,[\!]\, \mathcal{B}_c, \alpha)$$

then:

$$\mathcal{A} \sqsubseteq \mathcal{B}$$

Here, there is no data refinement involved and all actions and other internal actions are equivalent between the two systems. The assumptions ensure that $IT_{\{\mathcal{A}_a\}} \equiv IT_{\{\mathcal{B}_b, \mathcal{B}_c\}}$ since for any postcondition ϕ:

$$\begin{aligned}
& wp(IT_{\{\mathcal{B}_b, \mathcal{B}_c\}}) \\
& \equiv wp(\textbf{it}\ \mathcal{B}_b \parallel \mathcal{B}_c\ \textbf{ti}, \phi) \\
& \equiv wp(\textbf{it}\ \mathcal{A}_a\ \textbf{ti}, \phi) && \text{[By assumption]} \\
& \equiv wp(IT_{\{\mathcal{A}_a\}})
\end{aligned}$$

Hence the result follows from Property 3.

6.6 Refinement and security

One very important aspect of the deterministic security properties is that they are preserved by refinement. If action system $\mathcal{A} \setminus H$ is deterministic and $\mathcal{A} \sqsubseteq \mathcal{B}$ then $\mathcal{B} \setminus H$ is also deterministic. This follows by analogy with CSP since deterministic processes are maximal under the refinement order. The same is true for the lazy deterministic security property. Hence, any abstract system which can be shown to be secure with respect to a deterministic property may be refined to a secure implementation with no further security validation necessary.

6.7 Parallel refinement

Another technique introduced in [20] is the refinement of an action system by a set of action systems to be executed in parallel. For this, it must be possible to partition the state of the original system, with each disjoint subset forming the state of one of the parallel components. Thus parallel composition synchronises on common actions but no state variables are shared.

6.7.1 Examples

The first example shows how a single action system can be refined to a parallel implementation. Again, a variation of the simple key server is used.

Example 52

$[KEY]$
$STATUS ::= null \mid ready \mid sendA \mid sendB$

$KeyServer3 \mathrel{\widehat{=}}$

$$\left(\begin{array}{l}
\textbf{var } k : KEY;\ sess : STATUS \\
\textbf{initially } sess := null \\
\textbf{action } startsess\ :-\ sess = null \rightarrow sess := ready \\
\textbf{action } \textit{fixkey}\ :-\ sess = ready \rightarrow k :\in KEY;\ sess := sendA \\
\textbf{action } keytoA \textbf{ out } k! : KEY\ :- \\
\qquad sess = sendA \rightarrow k!, sess := k, sendB \\
\textbf{action } keytoB \textbf{ out } k! : KEY\ :- \\
\qquad sess = sendB \rightarrow k!, sess := k, null
\end{array}\right)$$

$KeyServer3$ establishes a key and distributes it to two users, A and B. The variable *sess* is used to maintain the correct flow of control and k represents the session key. A session is entered by selecting the action *startsess*. The key for this session is then determined by *fixkey*. The key is sent to the two users, first by *sendA* and then *sendB*.

The key server is to be refined by a system in which the server itself sends the key to user A only. User A is then responsible for sending the key on to B. Here the key server is modelled for the parallel version as $KeyServer4$,

and user A, *UserA*. Each has its own status variable and session key variable.

$KeyServer4 \mathrel{\widehat{=}}$

$$\left(\begin{array}{l}
\textbf{var } pk : KEY;\ psess : STATUS \\
\textbf{initially } psess := null \\
\textbf{action } startsess\ :-\ psess = null \rightarrow psess := ready \\
\textbf{action } fixkey\ :- \\
\qquad psess = ready \rightarrow pk :\in KEY;\ psess := sendA \\
\textbf{action } keytoA \textbf{ out } k! : KEY\ :- \\
\qquad psess = sendA \rightarrow k!, psess := pk, null
\end{array}\right)$$

$UserA \mathrel{\widehat{=}}$

$$\left(\begin{array}{l}
\textbf{var } Ak : KEY;\ Asess : STATUS \\
\textbf{initially } Asess := null \\
\textbf{action } startsess\ :-\ Asess = null \rightarrow Asess := sendA \\
\textbf{action } keytoA \textbf{ in } k? : KEY\ :- \\
\qquad Asess = sendA \rightarrow Ak, Asess := k?, sendB \\
\textbf{action } keytoB \textbf{ out } k! : KEY\ :- \\
\qquad Asess = sendB \rightarrow k!, Asess := Ak, null
\end{array}\right)$$

We now check to see that the parallel version really is a refinement of *KeyServer*3. The parallel system can be calculated as:

$$(KeyServer4 \;||\; UserA) \mathrel{\widehat{=}}$$

$$\left(\begin{array}{l}
\textbf{var } pk, Ak : KEY;\ psess, Asess : STATUS \\
\textbf{initially } psess, Asess := null, null \\
\textbf{action } startsess\ :- \\
\qquad psess = null \wedge Asess = null \rightarrow \\
\qquad\quad psess, Asess := ready, sendA \\
\textbf{action } fixkey\ :- psess = ready \rightarrow pk :\in KEY;\ psess := sendA \\
\textbf{action } keytoA \textbf{ out } k! : KEY\ :- \\
\qquad psess = sendA \wedge Asess = sendA \rightarrow \\
\qquad\quad k!, Ak, psess, Asess := pk, pk, null, sendB \\
\textbf{action } keytoB \textbf{ out } k! : KEY\ :- \\
\qquad Asess = sendB \rightarrow k!, Asess := Ak, null
\end{array}\right)$$

The retrieve relation, R, shows how the different possible abstract and concrete states are related:

$$\begin{array}{l}
R \mathrel{\widehat{=}} k = pk \wedge \\
\quad (\ (sess = null \wedge psess = null \wedge Asess = null) \vee \\
\quad\ (sess = ready \wedge psess = ready \wedge Asess = sendA) \vee \\
\quad\ (sess = sendA \wedge psess = sendA \wedge Asess = sendA) \vee \\
\quad\ (sess = sendB \wedge psess = null \wedge Asess = sendB \wedge Ak = k) \\
\quad\ )
\end{array}$$

The required proof of refinement is given in Appendix D. Since the *fixkey* operation represents the choice of a key by the server and requires no external participation, it would be reasonable to view it as an internal action. Both *KeyServer*3 and *KeyServer*4 could be altered to reflect this, and the refinement still holds. ♮

Example 53 Another possible use for parallel refinement is for the *MultiStack* specification of Section 5.5.1 where the stack for each classification *cl* could

be represented as:

$$Stack_{cl} \mathrel{\widehat=}$$

$$\left(\begin{array}{l}
\textbf{var } cstack_{cl} : \text{seq } DATA \\
\textbf{initially } cstack := \langle\rangle \\
\textbf{action } push \textbf{ in } u? : USER;\ sc? : CLASS;\ d? : DATA\ :- \\
\quad (clear\ u?) \leq sc? \rightarrow\ \ if\ sc? = cl \\
\qquad\qquad\qquad\qquad then\ cstack_{cl} := cstack_{cl}\hat{}\langle d?\rangle \\
\qquad\qquad\qquad\qquad else\ skip \\
\textbf{action } pop \textbf{ in } u? : USER;\ sc? : CLASS\ :- \\
\quad (clear\ u?) \leq sc? \rightarrow\ \ if\ cstack_{cl} \neq \langle\rangle \wedge sc? = cl \\
\qquad\qquad\qquad\qquad then\ cstack_{cl} := front\ cstack_{cl} \\
\qquad\qquad\qquad\qquad else\ skip \\
\textbf{action } top \textbf{ in } u? : USER;\ sc? : CLASS \textbf{ out } r! : REP\ :- \\
\quad true \rightarrow\ \ if\ sc? = cl \\
\qquad then\ (\ \ if\ (clear\ u?) < cl \\
\qquad\qquad\qquad then\ r! := error \\
\qquad\qquad\qquad else\ if\ cstack_{cl} = \langle\rangle \\
\qquad\qquad\qquad\qquad then\ r! := empty \\
\qquad\qquad\qquad\qquad else\ r! := Rep(last\ stack_{sc}) \\
\qquad\qquad)\ \ \ \

question whether the individual components of a secure parallel decomposition are secure. In fact, examples can easily be constructed to show that the components need not be secure.

Example 54 With:

$$\mathcal{B} \mathrel{\widehat{=}} \left(\begin{array}{l} \textbf{var } x, z : \{0,1\} \\ \textbf{initially } x, z := 0, 0 \\ \textbf{action } l1 \ :- \ x = 0 \rightarrow x, z := 1, 1 \\ \textbf{action } l2 \ :- \ x = 0 \rightarrow \textit{skip} \\ \textbf{action } h1 \ :- \ z = 1 \rightarrow \textit{skip} \end{array} \right)$$

$$\mathcal{C} \mathrel{\widehat{=}} \left(\begin{array}{l} \textbf{var } y : \{0,1\} \\ \textbf{initially } y := 0 \\ \textbf{action } l2 \ :- \ y = 1 \rightarrow \textit{skip} \\ \textbf{action } h1 \ :- \ y = 0 \rightarrow y := 1 \end{array} \right)$$

the parallel composition, $\mathcal{B} \parallel \mathcal{C}$, is:

$$\left(\begin{array}{l} \textbf{var } x, y, z : \{0,1\} \\ \textbf{initially } x, y, z := 0, 0, 0 \\ \textbf{action } l1 \ :- \ x = 0 \rightarrow x, z := 1, 1 \\ \textbf{action } l2 \ :- \ x = 0 \wedge y = 1 \rightarrow \textit{skip} \\ \textbf{action } h1 \ :- \ y = 0 \wedge z = 1 \rightarrow y := 1 \end{array} \right)$$

No trace of actions within the combined system can establish the guard of $l2$, and $l1$ is only enabled immediately after initialisation. Hence the system is eagerly and lazily secure. However, in $\mathcal{C}$, $l2$ is directly dependent on $h1$ and so $\mathcal{C}$ is not lazily secure. In $\mathcal{B}$, hiding $h1$ causes divergence, so this is not eagerly secure. ♮

The above example shows that a system can be secure but the individual components of its decomposition are not. This is a fact which must be bourne in mind during the development of secure systems. A secure action system may be refined to a parallel implementation with the knowledge that security is maintained by the interaction of the components. However, outside the

context of the original system no guarantees can be made for the security of the components. It would not be safe to take components from a secure development and place them in a different situation without checking the security implications of the new context.

The converse question can also be asked: that is, if $\mathcal{B}$ and $\mathcal{C}$ are both secure is $\mathcal{B} \,||\, \mathcal{C}$ also secure? Once again, the correspondence with CSP gives the answer. In [111] Roscoe and Wulf showed that if process P with alphabet A and process Q with alphabet B are lazily secure then so is $P \underset{A \cap B}{\|} Q$. Since action system parallel composition mirrors that for CSP, this is exactly what is needed. Hence, composing secure components yields a secure system.

Finally, suppose that $\mathcal{B} \,||\, \mathcal{C}$ is secure and that $\mathcal{B} \sqsubseteq \mathcal{D}$ and $\mathcal{C} \sqsubseteq \mathcal{E}$. Then, as demonstrated in [20], by the monotonicity of the parallel operator:

$$(\mathcal{B} \,||\, \mathcal{C}) \sqsubseteq (\mathcal{D} \,||\, \mathcal{E})$$

This allows components of a secure system to be refined individually. Chapter 2 noted various studies which have sought to set out the conditions under which components may be combined to preserve security. This section shows that deterministically secure action systems may be combined safely using parallel decomposition. For example, the system used by McCullough [81] to show the inadequacy of generalised noninterference for composability uses two subsystems $\mathcal{A}$ and $\mathcal{B}$ which are each shown to obey generalised noninterference individually. The action system equivalent of system $\mathcal{A}$ for instance is given in Figure 6.3.

This is insecure according to the deterministic definitions since $l2$ is directly affected by $h1$ and $h2$.

6.8 Related work

The simulation refinement method used here allows an integrated refinement of the events and the state variables of an action system. Data may be refined in a manner similar to that originally proposed by Hoare *et al.* [58] with the relationship between abstract and concrete variables specified in the retrieve

$$
\left(\begin{array}{l}
\textbf{var } count : \mathbb{N};\ stop : Boolean \\
\textbf{initially } count, stop := 0, false \\
\textbf{action } h1 \textbf{ in } n? : \mathbb{N} \ :- \\
\qquad stop = false \rightarrow count := count + 1 \\
\textbf{action } h2 \textbf{ out } n! : \mathbb{N} \ :- \\
\qquad stop = false \rightarrow count := count + 1 \\
\textbf{action } l1 \textbf{ out } m1! : Message \ :- \\
\qquad stop = false \rightarrow m1!, stop := \text{“}stop\_count\text{”}, true \\
\textbf{action } l2 \textbf{ out } m2! : Message \ :- \\
\qquad stop = true \rightarrow \ \textbf{if} \quad odd(count) \rightarrow m2! := \text{“}odd\text{”} \\
\qquad\qquad\qquad\qquad [\!] \quad even(count) \rightarrow m2! := \text{“}even\text{”} \\
\qquad\qquad\qquad\qquad \textbf{if}
\end{array}\right)
$$

Figure 6.3 Action system specification for the hook-up example

relation, R. Refinement of actions also maintains the CSP correspondence, preserving the inclusion of failures, divergences and infinite traces required for process refinement.

There are a number of related refinement techniques. Of particular interest is the Refinement Calculus [6, 94, 95, 96] where specification statements are given a weakest precondition semantics. A succession of correctness- preserving rules may be applied to statements in order to develop the abstract specification into an implementation. In an action system, actions may be given as specification statements. Refinement calculus techniques can then be used to refine the command within the framework of action system forwards simulation.

Another interesting comparison can be made between the method of simulation for action systems and refinement of action systems based on superposition [7]. This is seen as a special case of the general refinement calculus approach in which new variables may be added. New (auxilliary) actions may then be added to update the new variables. Old actions can be altered to include reference to the new variables, but they must preserve the effect of the original version on all existing variables. This technique is extended by Back and Sere [9] to allow for the presence of stuttering actions. Stuttering and stuttering equivalence were introduced by Lamport [73] and used [1] for

concurrent systems specified in a transition-axiom system based on temporal logic. They allow refinements to ignore the repetition of transitions in which the external state remains unchanged. A similar effect can be achieved for action systems by hiding stuttering actions. The refinement rule obtained for such a system [20, 123] is very similar to those of Back and Sere [9].

The overriding factor in favour of the approach used here is that it maintains the correspondence with CSP necessary for the definition of deterministic security properties. With Back's rule, for example, there is no distinction between internal and external choice. The CSP interpretation clearly distinguishes between a system with external choice such as:

$$\mathcal{A} \mathrel{\widehat{=}} \left(\begin{array}{l} \textbf{initially } \mathit{skip} \\ \textbf{action } h \ :- \ \mathit{skip} \\ \textbf{action } l \ :- \ \mathit{skip} \end{array} \right)$$

and one with internal (nondeterministic) choice:

$$\mathcal{B} \mathrel{\widehat{=}} \left(\begin{array}{l} \textbf{var } x : \mathit{Boolean} \\ \textbf{initially } x :\in \mathit{Boolean} \\ \textbf{action } h \ :- \ x \rightarrow x :\in \mathit{Boolean} \\ \textbf{action } l \ :- \ \neg\, x \rightarrow x :\in \mathit{Boolean} \end{array} \right)$$

$\mathcal{B}$ would not be allowed as a refinement of $\mathcal{A}$ because of the additional nondeterminism. Back's refinement approach basically ensures trace inclusion, and since the traces of the two systems are indistinguishable, $\mathcal{B}$ would be considered a refinement of $\mathcal{A}$. This would obviously be disastrous from the point of view of deterministic security.

6.9 Summary

Chapter 6 describes how action systems may be refined, with particuler reference to the effects on secure systems. Three basic refinement definitions were given, and from these may be derived other rules which can be used when particular conditions hold. The intention is that these will be simpler to apply. The examples presented here and in previous chapters have been

kept simple to introduce and explain the use of action systems. In the following chapters the use of action systems for the specification of more realistic systems is explored.

Chapter 7

Message passing in a network: case study 1

This chapter and the next consider how the methods described so far can be applied in practice. They investigate how the techniques of previous chapters can be used, questioning whether the use of action systems and of the security properties developed for action systems can support the development of secure systems in a more convenient and practical way than was possible with previous approaches. The emphasis here is on a pragmatic approach to the development of secure systems, with the basis for confidentiality provided by the deterministic conditions for action systems. However, the concern is wider than simply wishing to apply these conditions: the case studies of these two chapters are intended to explore the use of action systems in the wider context of the development of secure systems and to see whether the theoretically-appealing deterministic conditions really can be applied in practice. Chapter 7 is an exploration of these ideas, viewing the effects of different approaches on an existing action system specification. Chapter 8 provides a more focused case study based on the development of a kernel system.

The area of application addressed here is network security. This is a very broad area indeed, and has received much attention in recent years (see for example [70] and [41]). Networks in themselves provide a challenging area of study with many inter-related issues to be considered. The shift in computer usage away from isolated, stand-alone machines means that the

understanding of computer networks is a crucial aspect of computer science. Machines may be linked within an organisation to form a local area network (LAN) with cohesive security requirements applying across the range of components. However, a LAN may be connected to another LAN or to a wider network (such as the Internet) with widely differing security expectations and guarantees.

The widespread reliance on remotely-accessed machines and on the connections between them means that issues of security are of increasing concern. There have been many and varied attacks on networks (well-documented examples include the Internet Worm [124] and the widespread unauthorised access described by Stoll [126]). From the point of view of confidentiality, information may be at risk from eavesdropping whilst it is in transmission between two nodes of the system. The potential for unauthorised access on any node, both destination or intermediate, through which the information may pass should also be considered. The increasing demand for commercial transactions across networks has brought into focus many other aspects of security, including the integrity of data and the authentication of users.

For a network to be secure, all aspects of security must be considered across the range of interconnecting components. This is an extremely difficult task, but one weak point in the system could vitiate the most meticulous security arrangements elsewhere in the system. It is also important to consider the security implications of the interplay between different components of the system.

The specifications in this chapter concentrate on an abstract representation of a network, starting with a top-level view and gradually expanding to show how different security constraints can be added. This layered approach exploits the general specification technique of separation of concerns, allowing many details to be hidden at the higher levels. It is not possible here to address a comprehensive set of network security requirements (even if such could be defined) but a selection of possible approaches is investigated to show how each can be accommodated.

To describe a system fully a specification should be able to capture both

the functional and the security aspects of a system. This is where action systems can offer major benefits by supporting the description of both state changes and the succession of events within a network. Although it is possible to describe both functionality and confidentiality in a language such as CSP, the strength of action systems is their ability to describe the system in a state-rich notation and to allow the unified refinement of both state and events. Thus the approach described in previous chapters and applied in Chapters 7 and 8 has pragmatic advantages for the development of secure systems. The semantics of parallel action systems (described in Chapter 4) in which state is not shared between components is also well-suited to the description of networks.

The following section introduces the topic of computer networks, highlighting some of the main problems of security. The general specification of a network is then outlined, and this is used throughout the rest of the chapter as the framework within which security constraints may be specified and analysed.

7.1 Network security

In a computer network, many independent processors may be linked together in a variety of different ways. A message sent by one node may be routed via a number of intermediate nodes before it reaches its final destination. The message itself is generally broken into uniform pieces which can be transmitted in individual packets. The nodes of the network may be very diverse, with each component having different hardware and software which could cause significant communication problems. To allow for this, a layered protocol is generally used by each node of a network, providing a standard interface for network communication. The Open Systems Interconnection (OSI) Reference model as described in [60] sets out a model which recognizes seven steps that must be followed to allow seamless communication between applications running on different platforms. The importance of the layers is to partition the task of the communications protocol into different subtasks.

The number of layers is somewhat open to interpretation: the TCP/IP [26] is the protocol used for internet communication and this is based on a model which distinguishes only four main layers. However, the same basic layering pattern is used in both. It is necessary to mention the layered protocol architecture since this has a direct influence on the options available for secure communication.

Figure 7.1 shows the levels identified in the OSI model. An application on node A wishing to communicate with an application on node B sends the message through the layers of protocol. At each stage a different function is performed on the message as received from the previous layer. The effect is to package the message incrementally until it is ready for transmission. When received on node B, the corresponding layers are checked and the information added by node A is stripped off in reverse order. The layered protocols effectively establish communication between similar layers at each node. Figure 7.2 shows examples of the functions carried out at each level. This model will be of use in subsequent sections when considering how security constraints may be brought in.

7.2 A network specification

In [20] Butler presented an action system specification of a message passing system. This is a well-structured, layered specification which gives a broad abstract description at the top level and uses repeated levels of refinement to arrive at a representation of the system as the parallel composition of individual agents with the communication network. Here this is taken as an "off the peg" specification to form the basic network description in which to try out various security properties. It is useful to be able to reuse existing specifications, and one as robust as this provides solid ground on which to work. In this section the first two levels of the message-passing system are presented, tailored slightly for present purposes.

The set *Node* represents the finite set of nodes of the system and *Comm*

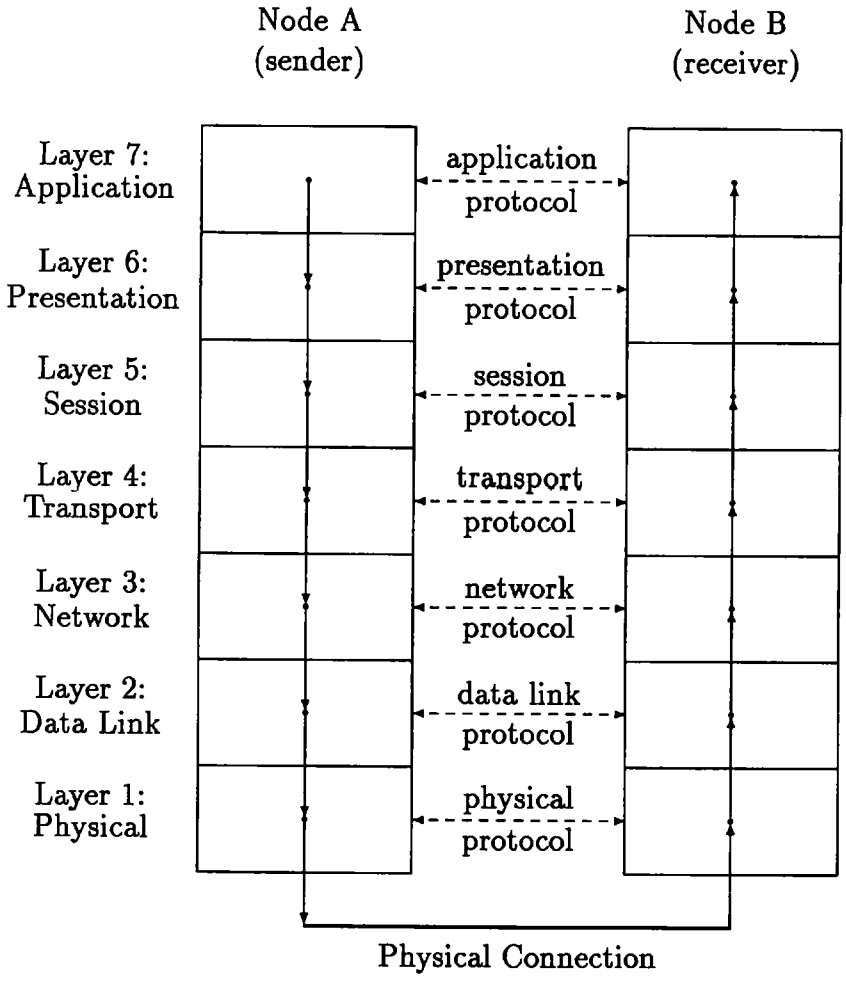

Figure 7.1 OSI model

OSI layer	Possible functions
Application	Makes network services available to user
Presentation	Transforms message to node-independent format
Session	Adds controls for synchronisation of dialogue
Transport	Orders packets with sequence numbers Reconstructs received packets to form original
Network	Computes paths and adds routing information
Data link	Organises communications across a single link Controls next (intermediate) destination of each packet
Physical	Sends/receives bit stream over physical link

Figure 7.2 Typical tasks carried out at each OSI level

the set of possible communications.

$$[Node, Comm]$$

An envelope consists of a destination node along with a message intended for that node:

$$Env \equiv Node \times Comm$$

A node may send a communication to any node in the system and can receive any communication sent to it. This is reflected in the top-level action system, $NET1$, where *comms* is a bag of envelopes which represents the full set of communications currently in the system. Here, **for** statements are used to represent sets of actions indexed by *Node* and actions are specified by specification statements (see Appendix B).

$$NET1 \mathrel{\widehat{=}}$$

$$\left(\begin{array}{l}
\textbf{var } comms : \text{bag } Env \\
\textbf{initially } comms : [comms = \mathbf{[\!]}] \\
\textbf{for } s \in Node \textbf{ action } send_s \textbf{ in } (r?, m?) : Env \ :- \\
\quad comms : [comms = comms_0 + \mathbf{[\!}(r?, m?)\mathbf{\!]}] \\
\textbf{for } r \in Node \textbf{ action } receive_r \textbf{ out } m! : Comm \ :- \\
\quad comms : [(r, m!) \text{ in } comms_0 \wedge comms = comms_0 - \mathbf{[\!}(r, m!)\mathbf{\!]}]
\end{array}\right)$$

Initially, *comms* is empty. Each node may send a communication to any node in the system, and may also receive messages sent to it. The next level of refinement reveals more of the internal structure of the system, with internal actions to forward messages along links of the network. At this stage, questions of routing must be addressed and a brief summary of the route description defined in [20] is given.

A network is a fixed set of pairs, showing the connections between nodes:

$$net : Node \leftrightarrow Node$$

Routes of the network are defined to be subrelations of the net such that for any route R and for any nodes a, b and c, if:

$$(a \ R \ b) \wedge (b \ R^* \ c)$$

then:

$$e_R(b, c) < e_R(a, c)$$

where R^* is the reflexive transitive closure of R and $e_R(x, y)$ is the length of the maximum path (containing no repeated nodes) from x to y in R. This ensures that following a route brings a communication step by step to its destination. Tagging the routes of the network allows routing to be dealt with in the specification without detailing the actual routes:

$$route : Tag \rightarrow Routes(net)$$

It is assumed that there is at least one route linking each pair of nodes (this is formalised in [20]).

This allows a refinement of $NET1$ in which each node may hold messages for forwarding. Communication is modelled by link actions which transfer messages between nodes. This is regarded as internal network behaviour and so link actions are hidden within the action system. Communications are assigned an appropriate route and marked with the corresponding tag. During transmission, they may be held in the stores of individual nodes or they may be on a link between nodes. The following Z schema can be used to represent the state:

NS2

$$store : Node \rightarrow \text{bag}(Tag \times Env)$$
$$link : net \rightarrow \text{bag}(Tag \times Env)$$

$$\forall i : Tag;\ r : Node;\ m : Comm;\ a, b : Node \bullet$$
$$(i, (r, m)) \text{ in } store(a) \Rightarrow (a, r) \in route(i)^* \wedge$$
$$(a, b) \text{ in } net \wedge (i, (r, m)) \in link(a, b) \Rightarrow (b, r) \in route(i)^*$$

Whether a communication is at a node or on a link it must still be on course within its specified route. Action system $NET2$ in Figure 7.3 specifies network behaviour at this level. A communication is stored at a node and then forwarded by internal links through intermediate nodes on its route until it

$NET2 \triangleq$

$$
\left(\begin{array}{l}
\textbf{var } NS2 \\
\textbf{initially } \begin{matrix} store, \\ link \end{matrix} : \left[\begin{array}{l} (\forall\, a : Node \bullet Node(a) = [\![]\!]) \wedge \\ (\forall (a,b) : net \bullet link(a,b) = [\![]\!]) \end{array}\right] \\
\textbf{for } s \in Node \textbf{ action } send_s \textbf{ in } (r?, m?) : Env \ :- \\
\qquad store \ : \left[\begin{array}{l} (\exists\, i : Tag \bullet (s, r?) \in route(i)^* \wedge \\ \qquad store(s) = store_0(s) + [\![(i,(r?,m?))]\!]) \end{array}\right] \\
\textbf{for } r \in Node \textbf{ action } receive_r \textbf{ out } m! : Comm \ :- \\
\qquad store \ : \left[\begin{array}{l} (\exists\, i : Tag \bullet (i,(r,m!)) \text{ in } store_0(r) \wedge \\ \qquad store(r) = store_0(r) - [\![(i,(r,m!))]\!]) \end{array}\right] \\
\textbf{internal } forward \ :- \\
\qquad \begin{matrix} store, \\ link \end{matrix} : \left[\begin{array}{l} (\exists\, a,b,r : Node;\ i : Tag;\ m : Comm \bullet \\ \qquad (i,(r,m)) \text{ in } store_0(a) \wedge r \neq a \wedge \\ \qquad (a,b) \in route(i) \wedge (b,r) \in route(i)^* \wedge \\ \qquad store(a) = store_0(a) - [\![(i,(r,m))]\!] \wedge \\ \qquad link(a,b) = link_0(a,b) + [\![(i,(r,m))]\!]) \end{array}\right] \\
\textbf{internal } relay \ :- \\
\qquad \begin{matrix} store, \\ link \end{matrix} : \left[\begin{array}{l} (\exists\, a,b,r : Node;\ i : Tag;\ m : Comm \bullet \\ \qquad (i,(r,m)) \text{ in } link_0(a,b) \wedge \\ \qquad link(a,b) = link_0(a,b) - [\![(i,(r,m))]\!]) \wedge \\ \qquad store(b) = store_0(b) + [\![(i,(r,m))]\!] \end{array}\right]
\end{array}\right)
$$

Figure 7.3 Second level network specification

is received at its destination node. Eventual receipt of messages is ensured by the decrease in length of maximum paths at each step. That this is a refinement of the original:

$$NET1 \sqsubseteq NET2$$

is proved by Butler [20] using the retrieve relation:

$$\begin{aligned} &NS2 \wedge \\ &comms = (\Sigma a \in Node \bullet untag(store(a))) \\ &\qquad + (\Sigma(a,b) \in net \bullet untag(link(a,b))) \end{aligned}$$

where *untag* is the function which constructs the bag of untagged envelopes from the corresponding bag of tagged ones.

7.3 Insecure nodes

Given a basic view of the network there are many ways in which security constraints can be added. First, the case in which each message may be given a security classification reflecting its sensitivity is considered. Nodes will also be assigned a classification representing the maximum security level of material they are allowed to process. As with the network description, the clearance function is a system constant:

$$clear : Node \rightarrow Class$$

where *Class* is a linearly ordered set of security classifications adhered to throughout the network. An envelope must now state the classification of the communication, which leads to the definition:

$$SecEnv \mathrel{\widehat=} Node \times Comm \times Class$$

Then for any envelope e:

$$node\ e, \quad comm\ e, \quad class\ e$$

project out the associated recipient, message and classification respectively. The security policy represented here is a very simple-minded one and makes

$C1NS1$

$comms$: bag $SecEnv$

$\forall c : comms \bullet clear\ (node\ c) \geq (class\ c)$

Figure 7.4 The schema $C1NS1$

a number of assumptions which would not be justified in most systems. This is taken as a starting point to show how the ideas of previous chapters can be applied. The later sections of this chapter clarify and question some of the assumptions and modify the specification accordingly.

The state of the system specifies that no communication may be sent to a node of insufficient clearance. This is given by the schema $C1NS1$ defined in Figure 7.4. At the top level it is necessary to specify that a message can only be sent to a suitably cleared recipient (no read up) and that the classification of the message must be at least as great as the sender (no write down). The classification is included as an output with the received message so that it may be handled appropriately at the receiving node. The system is shown in Figure 7.5. The guards of a specification statement $x : [post]$ may be calculated explicitly as $(\exists x \bullet post)[x/x_0]$ (see Appendix B):

$$gd\ send_s \equiv true$$
$$gd\ receive_r \equiv (r, m!, c!) \in comms$$

This system may appear at first glance to maintain security, but checking the nondeterministic security properties quickly reveals a problem. As outlined in Section 4.3.2, when dealing with value-passing action systems the *commgd* must be calculated to give a true picture of the possible refusals. In the case of $receive_r$, for any set S of outputs:

$$commgd(receive_r.S) = (gd\ receive_r) \wedge wp(receive_r, (m!, c!) \in S)$$

This has the effect of adding the requirement that the set of messages for r in *comms* is a subset of S. However, if the two distinct messages $m1$ with classification $c1$ and $m2$ with classification $c2$ are the messages in the system

$$
C1NET1 \mathrel{\widehat{=}}
$$

$$
\left(\begin{array}{l}
\textbf{var } C1NS1 \\
\textbf{initially } comms : [comms = [\!]] \\
\textbf{for } s \in Node \textbf{ action } send_s \textbf{ in } (r?, m?, c?) : SecEnv \ :- \\
\qquad comms \ : \left[\begin{array}{l}
(clear\ s \leq c? \wedge c? \leq clear\ r? \wedge \\
\qquad comms = comms_0 + [\!|(r?, m?, c?)|\!]) \\
\vee \\
(\neg\ (clear\ s \leq c? \wedge c? \leq clear\ r?) \wedge \\
\qquad comms = comms_0)
\end{array}\right] \\
\textbf{for } r \in Node \textbf{ action } receive_r \textbf{ out } m! : Comm;\ c! : Class \ :- \\
\qquad comms \ : \left[\begin{array}{l}
(r, m!, c!) \text{ in } comms_0 \wedge \\
comms = comms_0 - [\!|(r, m!, c!)|\!]
\end{array}\right]
\end{array}\right)
$$

Figure 7.5 Top level network specification with security levels

for recipient r, then:

$$
\begin{aligned}
& commgd(receive_r.\{(m1, c1)\}) \\
& \equiv (r, m1, c1) \text{ in } comms \wedge wp(receive_r, (m!, c!) \text{ in } \{(m1, c1)\}) \\
& \equiv true \wedge wp(receive_r, (m!, c!) \text{ in } \{(m1, c1)\}) \\
& \equiv \{(m1, c1), (m2, c2)\} \subseteq \{(m1, c1)\}) \\
& \equiv false
\end{aligned}
$$

So, for distinct messages, r could refuse to output each specific message (but not both) in this situation, although either is possible.

In terms of system behaviour, the problem is that there is a nondeterministic choice between outputs. This could in theory be resolved by the influence of high-level actions and a covert channel established through the output order of low-level messages.

The deterministic security property has done its job in detecting a possible covert channel. The system developer now has to decide how to act on this information. One approach might be to decide that, if this is the worst the system can do we are prepared to live with it. This could be a valid approach depending on how stringent the security requirements for the

system are. Another approach would be to alter the specification to remove the nondeterminism. The problem with this is that it means including a lot of otherwise unnecessary detail at the top level and losing the clarity of the specification. All that is required is a mechanism to resolve output order. This is a common situation in specifications. It is similar to McLean's example of a secure stack [87] where, for instance, the action to be taken by the system if an attempt is made to POP or TOP an empty stack is left undefined. McLean's theory solves this problem by requiring refinements to resolve the nondeterminism in a benign way. It takes the onus away from the specification, but it introduces non-standard requirements into refinement rules.

For the purpose of this discussion the choice is made to add a mechanism for selecting the next message to output without fully defining how this is done. For each node r:

$$next_r : \text{bag } Comm \nrightarrow Comm$$

where $next_r$ picks the next message from any non-empty bag. The way in which the "next" message is decided is not allowed to depend on any high-level actions. This is not an entirely satisfactory solution since the ordering of outputs is very limiting with no possibility of altering the order in future refinements. Parallel decomposition becomes very awkward in such a rigid structure. This problem is inherent in the deterministic approach. It is encountered again and discussed further in the next chapter. For now, it is noted that a potential covert channel has been flagged and $next_r$ is introduced in order to pursue the specification further. With this the action $receive_r$ can be amended to:

$$comms : \left[\begin{array}{l} (r, m!, c!) \text{ in } comms_0 \wedge \\ (r, m!, c!) = next_r(comms_0) \wedge \\ comms = comms_0 - \llbag (r, m!, c!) \rrbag \end{array} \right]$$

Now, for any output message m at classification c:

$$commgd(receive_r.\{(m, c)\})$$
$$\equiv (r, m, c) \in comms \wedge (r, m, c) = next_r(comms)$$

There is now no choice of message to be output, so the action system cannot refuse an output in which it might also engage. To ensure the security of the amended system it is necessary to show that:

$obs_H(C1NET1)$ deterministic

It is sufficient to choose an arbitrary classification, c, to divide the actions between high and low levels: H actions are all those with $clear(node) > c$, with the node being the s of a *send* or the r of a *retrieve*. The guard for any *send* action is always *true*. The *commgd* for a low-level *receive* is as above. Only r can receive r's messages, and r now receives messages in a fixed order, so it could only be affected by H actions if a *send* from a node of higher clearance puts some message, m, into the system for r where the classification of m is greater than $clear(r)$. However, the *send* action demands that the classification of the message be less than or equal to the clearance of r.

This informal description gives an outline for a proof. However, a formal proof of determinism even for such a small system is complicated by the need to prove the property for all traces. In the next chapter a technique is introduced to make formal proofs of determinism more manageable.

7.4 Refining the basic specification

Once a basic top-level specification is shown to be secure then any refinement of that specification is also guaranteed to be secure with no further proof necessary (apart from that of the refinement itself). The action system of the previous section may be refined to a version of $NET2$ with security conditions. We assume here that a message may not pass through nodes whose classification is less than that of the message. The function:

$rclear : Routes(net) \rightarrow Class$

maps each route in the network to the maximum clearance of traffic it is allowed to deal with. It is assumed that at least one route exists for every

possible node to node communication within the system. This abstracts away from the details of how the clearance would be determined. It is a very crude way of assessing security since a route may encompass many more nodes than those which any given message will actually pass through. However, it provides a first-pass abstraction to which detail can be added at a later stage.

The state is the same as for $NET2$ except that envelopes now contain classified messages.

$C1NS2$

$$store : Node \rightarrow \text{bag}(Tag \times SecEnv)$$
$$link : net \rightarrow \text{bag}(Tag \times SecEnv)$$

$$\forall i : Tag;\ r : Node;\ m : Comm;\ c : Class;\ a, b : Node \bullet$$
$$(i, (r, m, c)) \text{ in } store(a) \Rightarrow (a, r) \in route(i)^*$$
$$(a, b) \in net \wedge (i, (r, m, c)) \text{ in } link(a, b) \Rightarrow$$
$$(b, r) \in route(i)^*$$

The system with internal actions is given in Figure 7.6. Here:

$$next(store_0(r)) = next_r(comms_0)$$

As before:

$$C1NET1 \sqsubseteq C1NET2$$

with the refinement relation:

$$C1NS2 \wedge$$
$$comms = (\Sigma a \in Node \bullet untag(store(a)))$$
$$+(\Sigma(a, b) \in net \bullet untag(link(a, b)))$$

The proof of this refinement follows a similar pattern to that of the corresponding level in [20].

$C1NET2 \triangleq$

$$
\left(\begin{array}{l}
\textbf{var } C1NS2 \\
\textbf{initially } \begin{array}{l} store, \\ link \end{array} : \left[\begin{array}{l} (\forall\, a : Node \bullet Node(a) = [\!]) \wedge \\ (\forall (a,b) : net \bullet link(a,b) = [\!]) \end{array}\right] \\
\textbf{for } s \in Node \textbf{ action } send_s \textbf{ in} \\
\qquad r? : Node;\ m? : Comm;\ c? : Class\ :- \\
\qquad store\ : \left[\begin{array}{l} (clear\ s \leq c? \wedge c? \leq clear\ r? \wedge \\ \quad (\exists\, i : Tag \bullet (s, r?) \in route(i)^* \wedge \\ \qquad rclear(route\ i) \geq c? \wedge \\ \qquad store(s) = store_0(s) + [\!](i,(r?,m?,c?))[\!]) \\ ) \\ \vee \\ (\neg\ (clear\ s \leq c? \wedge c? \leq clear\ r?) \wedge \\ \quad store = store_0) \end{array}\right] \\
\textbf{for } r \in Node \textbf{ action } receive_r \textbf{ out } m! : Comm;\ c! : Class\ :- \\
\qquad store\ : \left[\begin{array}{l} (\exists\, i : Tag \bullet (i,(r,m!,c!)) \text{ in } store_0(r) \wedge \\ \qquad (r,m!,c!) = next(store_0(r)) \wedge \\ \qquad store(r) = store_0(r) - [\!](i,(r,m!,c!))[\!]) \end{array}\right] \\
\textbf{internal } forward\ :- \\
\qquad \begin{array}{l} store, \\ link \end{array} : \left[\begin{array}{l} (\exists\, a,b,r : Node;\ i : Tag;\ m : Comm;\ c : Class \bullet \\ \qquad (i,(r,m,c)) \text{ in } store_0(a) \wedge r \neq a \wedge \\ \qquad (a,b) \in route(i) \wedge (b,r) \in route(i)^* \wedge \\ \qquad store(a) = store_0(a) - [\!](i,(r,m,c))[\!] \wedge \\ \qquad link(a,b) = link_0(a,b) + [\!](i,(r,m,c))[\!]) \end{array}\right] \\
\textbf{internal } relay\ :- \\
\qquad \begin{array}{l} store, \\ link \end{array} : \left[\begin{array}{l} (\exists\, a,b,r : Node;\ i : Tag;\ m : Comm;\ c : Class \bullet \\ \qquad (i,(r,m)) \text{ in } link_0(a,b) \wedge \\ \qquad link(a,b) = link_0(a,b) - [\!](i,(r,m,c))[\!]) \wedge \\ \qquad store(b) = store_0(b) + [\!](i,(r,m,c))[\!] \end{array}\right]
\end{array}\right)
$$

Figure 7.6 Second level specification - classified messages

7.5 A different view of system security

Since $C1NET1$ is secure and $C1NET1 \sqsubseteq C1NET2$ it follows that $C1NET2$ is also secure. However, there is a problem with modelling the system this way. Since *forward* and *relay* are internal actions they play no part in what can be revealed by the system, other than in their immediate role of passing communications on to the rightful recipient. In the specification of $C1NET2$ care was taken to avoid communications passing through insecure nodes, and the route for any communication was constrained accordingly. However, the refinement proof makes no use of this constraint: the refinement would still hold if messages were allowed to follow an insecure path. This is because there is no representation in the specification of external behaviour at intermediate nodes.

One way to address this is to model the potential leak situations explicitly. This is the approach taken by Roscoe [106] and Schneider [119] who both model the behaviour of an intruder to the system. To do this, the possible interactions of the enemy with the system must be set out. Here, the concern is with confidentiality, so the enemy's ability to gain information from the system must be represented. For example, in the network specification it might be assumed that a node with clearance c could not be trusted to handle material of higher classification. So possible leaks might be modelled as outputs of any communication with classification greater than c passing through that node. The success of this approach depends on the developer's ability to imagine and characterise the behaviour of the intruder. Thus it is important to analyse the possible threats to the system, and to capture this as part of the system specification. This cannot be done at the top level because the structure of stores is not represented there. It is useful to be able to construct a separate action system to represent the intruder's behaviour. This can then be placed in parallel with the network specification (with additional actions representing the leakage of information to correspond with the intruder's actions). To check system security it is then necessary to determine whether the interface of the system with the intruder is deterministic.

There is another deficiency with the model, because no account has been

taken of the need to protect communications whilst being transmitted on a link. Rather than assigning different levels of security to a link, it is more usual to assume that a link can either be tapped into or that it cannot, that is: it is either secure or it is insecure. In a network this may well be the case, for instance, with links within a local area network considered secure and links outside considered insecure. Communication over an external link can only be relied upon as secure if the message is encrypted in a way which an eavesdropper is considered to be unable to decipher in a timely way. Thus for the next attempt at network specification, encryption is introduced.

At the beginning of this chapter an outline was given of the layered architecture for network communication. If messages are to be encrypted (and later decrypted) a number of questions arise. For example, what form of encryption should be used, how it should be specified and what architectural layer should perform the task. The two most common levels at which cryptographic tasks are carried out are the physical (or data link) level and, higher up, at the application level. A message encrypted at the lower level is encrypted for a single link only. The corresponding physical (or data link) layer at the receiving end of the link must decipher the communication to find the header information which determines the forward routing of the message. The message is again encrypted and sent on the chosen link. This method is known as link encryption. It has the advantage of being easily implementable at the hardware level and is transparent to the applications which are communicating. However, since encryption is carried out at each node, the message will appear in plaintext on each node. In contrast, encrypting at the application level allows header information to be packaged around the encrypted message, which can then be sent in encrypted form right through to its final destination with no need for access to be gained by intermediate nodes. However, if header information is not encrypted, anyone intercepting the message may discover the intended recipient and other information. This information could possibly be exploited by traffic analysis or used for attacks involving replay of messages.

Whilst link encryption and end-to-end encryption are the two models

most frequently cited, encryption at other levels may also be used. For example, end- system level encryption is carried out, not by individual applications, but by the end-system (usually a single machine) which hosts them. It provides protection on an end-system to end-system basis and would be carried out at the transport or sub- network independent layers. This approach has some of the advantages of end-to-end encryption but it cannot provide application-specific security. However, it can be used if a single (consistent) security policy is sufficient for a particular node. It can operate more efficiently and it ensures that some of the layered header information is encrypted with the message.

Link encryption is specified in the next section, followed by end system encryption in Section 7.7. End-to-end encryption is addressed in Section 7.8.

Encrypted messages can be problematic in the modelling of secure systems. Usually, high-level behaviour is not allowed to influence the view of a low-level user. However, encrypted high-level messages may be seen by low-level users and it is assumed that, without knowledge of the key, the message itself cannot be derived. Of course, there may still be other information which even an encrypted communication reveals: the fields of an unencrypted header may be read and the very presence of the message can contribute to intelligence gained through traffic analysis. However, for present purposes the situation is simplified by assuming that an encrypted communication yields no information unless the key is also available.

7.5.1 Introducing encryption

The format of messages must be altered to allow for the possibility of encryption. Here the situation is kept as simple as possible by distinguishing between plaintext and keys:

$$[PLAINTEXT, KEY]$$

A message may consist of either of these, or it may be the result of encrypting another message using a key:

$$\begin{aligned} Mess ::= \quad & text\langle\langle PLAINTEXT\rangle\rangle \\ | \quad & key\langle\langle KEY\rangle\rangle \\ | \quad & enc\langle\langle KEY \times Mess\rangle\rangle \end{aligned}$$

Notice that this definition allows multiple layers of encryption. An envelope once again consists of a message and the recipient node. This time however a classification is not included. It is assumed at this stage that all communications within the system are to be encrypted, returning later to consider other possibilities. For now, an envelope is defined:

$$Env \mathrel{\widehat=} Node \times Mess$$

Encryption may be carried out on a message but (for link encryption) a message already tagged with additional information, such as routing, may also be encrypted. Thus a communication as transmitted across the medium is represented as follows:

$$\begin{aligned} & TCOM \mathrel{\widehat=} Tag \times Env \\ & ECOM \mathrel{\widehat=} KEY \times TCOM \\ & COMM ::= \quad tcom\langle\langle TCOM\rangle\rangle \\ & \qquad\quad | \; ecom\langle\langle ECOM\rangle\rangle \end{aligned}$$

It is assumed that a public key encryption system is used and that for each node n in the network, pk_n is the public key for that node and sk_n is its secret key. Again, this represents just one possible situation. Further variations could be given for secret key systems or for networks which use both.

7.5.2 The information gained by an intruder

The shift in approach to modelling described in this section is greater than might at first be apparent. With a system-wide convention of classifications it is easy to characterise unwanted activity as anything causing information flow from high to low levels. In such a case it is clear how the deterministic

security conditions may be applied to check the appearance of the system to a lower-level user. However, the realities of communications security lead to the consideration of networks with insecure connections where the major tool for protecting confidentiality is encryption. It would be the intention that an encrypted message should be decipherable only by the recipient for whom it is intended. However, it is not necessarily the case that an encrypted message should not be viewed by other users - indeed, it is the very purpose of encryption to allow communications to pass over insecure media without conveying confidential information to uncleared parties. This leads to the view that encrypted messages should somehow be an "exception" to the information flow assessment. On the other hand, it would not be correct to dismiss encrypted messages as having no effect on system security. Amongst the ways they can convey information are:

- the presence of messages between certain destinations
- unencrypted information in the message header
- compromise of keys allowing unauthorised decryption
- weak encryption process vulnerable to cryptographic attack.

The choice of whether or not these are to be examined is a part of the analysis of the system. For instance, in all the examples here using encryption it will be assumed that decryption without knowledge of the relevant key is not within the scope of possible intruder activity. This places the security of a system within certain limits, such as "secure assuming encryption cannot be broken". It is very useful to be able to do this since it allows the analysis to concentrate on the specific aspects which are important for any given approach. However, it is also important to recognise that such simplifications are being made and to note any implicit assumptions.

As mentioned above, each of the following approaches models the behaviour of an intruder. This will necessarily constrain the consideration of security. The network specification itself must be allowed to include the corresponding actions representing participation of the system in information-

leaking actions. For instance, information passing along a link may be intercepted by the intruder and this should be reflected in the specification both of the medium and of the eavesdropper. Examples of how this is done are given in the following specifications.

Once the interaction of an intruder has been defined for a system, the deterministic security properties may be checked as before.

7.6 Link encryption

The state of the network is still basically the same as in the previous example, but the structure of the underlying sets has been changed somewhat. Each node continues to deal with tagged envelopes, but these will be forwarded as encrypted communications. This is given by the schema $C2NS2$.

$C2NS2$

$$store : Node \rightarrow \text{bag}(Tag \times Env)$$
$$link : net \rightarrow \text{bag}\ COMM$$

$$\forall i : Tag;\ r, a, b : Node;\ m : Mess;\ k : KEY \bullet$$
$$(i, (r, m)) \text{ in } store(a) \Rightarrow (a, r) \in route(i)^* \wedge$$
$$(a, b) \in net \wedge tcom(i, (r, m)) \text{ in } link(a, b) \Rightarrow$$
$$(b, r) \in route(i)^* \wedge$$
$$(a, b) \in net \wedge ecom(k, (i, (r, m))) \text{ in } link(a, b) \Rightarrow$$
$$(b, r) \in route(i)^*$$

The system with link encryption is then represented by the action system $C2NET2$ in Figure 7.7. The action *tap* represents the way in which an intruder might gain information from the system. Any message sent across a link may be overheard. However, we make the assumption that only unencrypted messages can convey information to the intruder. As discussed in the previous section, this assumes that no encrypted message can be deciphered by the intruder. Of course, even if the encryption algorithm is assumed to be unbreakable it would still be possible for the eavesdropper to decrypt

$C2NET2 \mathrel{\widehat{=}}$

$$
\left(\begin{array}{l}
\textbf{var } C2NS2 \\
\textbf{initially } \begin{array}{l} store, \\ link \end{array} : \left[\begin{array}{l} (\forall\, a : Node \bullet store(a) = [\!]) \wedge \\ (\forall (a,b) : net \bullet link(a,b) = [\!]) \end{array}\right] \\
\textbf{for } s \in Node \textbf{ action } send_s \textbf{ in } r? : Node;\ m? : Mess\ :- \\
\qquad store\ : \left[\begin{array}{l} \exists\, i : Tag \bullet (s, r?) \in route(i)^* \wedge \\ \qquad store(s) = store_0(s) + [\!](i,(r?,m?))[\!] \end{array}\right] \\
\textbf{for } r \in Node \textbf{ action } receive_r \textbf{ out } m! : Mess\ :- \\
\qquad store\ : \left[\begin{array}{l} \exists\, i : Tag \bullet (i,(r,m!)) \text{ in } store_0(r) \wedge \\ \qquad store(r) = store_0(r) - [\!](i,(r,m!))[\!] \end{array}\right] \\
\textbf{internal } forward\ :- \\
\qquad \begin{array}{l} store, \\ link \end{array} : \left[\begin{array}{l} \exists\, a,b,r : Node;\ i : Tag;\ m : Mess \bullet r \neq a \wedge \\ \qquad (a,b) \in net \wedge (i,(r,m)) \text{ in } store_0(a) \wedge \\ \qquad (a,b) \in route(i) \wedge (b,r) \in route(i)^* \wedge \\ \qquad store(a) = store_0(a) - [\!](i,(r,m))[\!] \wedge \\ \qquad link(a,b) = link_0(a,b) + \\ \qquad\qquad [\!]ecom(pk_b,(i,(r,m)))[\!] \end{array}\right] \\
\textbf{internal } relay\ :- \\
\qquad \begin{array}{l} store, \\ link \end{array} : \left[\begin{array}{l} \exists\, a,b : Node;\ i : Tag;\ c : TCOM \bullet \\ \qquad (a,b) \in net \wedge ecom(pk_b,c) \text{ in } link_0(a,b) \wedge \\ \qquad link(a,b) = link_0(a,b) - [\!]ecom(pk_b,c)[\!] \wedge \\ \qquad store(b) = store_0(b) + [\!]c[\!] \end{array}\right] \\
\textbf{action } tap \textbf{ out } c! : COMM\ :- \\
\qquad\qquad : \left[\ \exists\, a,b : Node \bullet c! \text{ in } link(a,b)\ \right]
\end{array}\right)
$$

Figure 7.7 System with link encryption

messages if keys have been compromised. For present purposes we will assume that the only way for keys to be leaked is within the system (that is, we discount at this stage the possibility of keys being passed, for example, by an informant who simply tells the eavesdropper the secret keys over the telephone). In this case, a key can only be leaked if it is sent as plaintext (otherwise knowledge of a prior key would be needed) so it is sufficient to check that no unencrypted messages can be overheard.

The actions of the intruder are kept as simple as possible here, but it is equally possible to imagine a more sophisticated interaction. For example, each communication is seen in isolation here but it would be possible to investigate the effects of accumulated knowledge by maintaining a state variable of the collection of information the user has gathered so far. An approach along these lines is given by Scheider [119] where a function is used to describe how information may be derived from communications (for instance, an encrypted message plus the relevant key yield the plaintext message). Any newly intercepted communication may be combined with the existing set and all new information generated can be output to the user. This type of interaction is readily accommodated in action systems since the state variables are all dealt with explicitly. It allows an analysis to be carried out of what information is compromised following a particular leak: for example, if a certain key is compromised it may be useful to find out what further losses are sustained throughout the system.

7.6.1 Security of the link encryption system

To establish whether the specification is secure we view $\{tap\}$ as the allowed set of actions of the uncleared user and $\{send_s, receive_r\}$ as the "high- level" actions in the system. The condition to be checked is:

$$obs_{\{send_s, receiver_r\}}(C2NET2) \text{ deterministic}$$

However, in this interpretation any message entering the system could affect the view of the intruder even if the message were encrypted. This interference is certainly worth noting, but we wish to discount the influence of encrypted

messages for now. In effect, actions $tap.(ecom\ e)$ for any e need to be excluded from the interface of the low level user when assessing security. What is required is a way of stating that, in the combined system with *send* and *receive* obscured, no intruder action other than one of these $tap.(ecom\ e)$ may be nondeterministically enabled or disabled. The $tap.(ecom\ e)$ should not be obscured in the system—they are not high-level events and their enablement and the effects of their execution are apparent to the intruder. To represent the situation an approach is needed which allows security to be assessed for limited aspects of the low-level user's view of the system. In the following sections a possible way of doing this is suggested and its consequences explored for the link encryption system.

7.6.2 Limited security for the link encryption system

The following definition provides a way to assess security with respect to particular low level actions.

Definition 38 *Suppose $\mathcal{A}$ is an action system with alphabet A and with $B \subseteq A$. $\mathcal{A}$ is deterministic within B if $divs(\mathcal{A}) = \emptyset$ and for each trace tr and each action $x \in B$:*

$$\overline{wp}(\mathcal{A}_{\langle init \rangle^\frown tr}, gd(\mathcal{A}_x)) = wp(\mathcal{A}_{\langle init \rangle^\frown tr}, gd(\mathcal{A}_x))$$

We can then say that a system $\mathcal{A}$ is lazily secure for high level actions H within the set of low level actions $L' \subseteq L$ if:

$obs_H(\mathcal{A})$ is deterministic within L'.

This reduces to the original definition if $L' = L$. Definition 38 extends in a natural way to systems with internal actions and value-passing systems. It allows the security of a system to be treated within clearly stated boundaries. As mentioned above, it is important that these limitations are understood and suitable justification given for ignoring other interaction. In the present example, discounting encrypted outputs certainly allows the eavesdropper to be influenced by the sending node, but it is an interaction we have declared

ourselves willing to accept in this example (as would be the case in most real life situations). However, it should not be forgotten that this gives the potential for a covert channel in the system. If that risk is not acceptable, then preventative measures must be taken.

For the network specification of Figure 7.7 it is necessary to check that :

$$obs_{\{send,receive\}}(C2NET2) \text{ is deterministic within } \{tap.c \mid c \in \operatorname{ran} tcom\}$$

For any $t : TCOM$ the guard of action $tap.(tcom\ t)$ is:

$$(\exists\, a, b : Node \bullet (tcom\ t) \text{ in } link(a, b))$$

So, as would be expected, for this action to be enabled the intercepted communication must be the unencrypted communication $(tcom\ t)$. No trace of actions in the system can reach a state where this guard is enabled. This is a result that can be proved formally using structural induction over traces. Informally it can be seen that this property is a trivial consequence of the fact that no unencrypted messages are ever placed on a link. So $tap.(tcom\ t)$ can never be enabled and hence by Theorem 6 the system is secure for unencrypted communications.

7.6.3 Refinement of the limited security approach

One of the main advantages of the deterministic security approach is the way in which the properties are preserved by refinement. It should also be questioned whether the restricted interpretation of security also has this property. Suppose $\mathcal{A}$ is an action system which is lazily secure with respect to H within the set of low level actions $L' \subseteq L$. Suppose that in a refinement of this the concrete system $tr_c \hat{}\langle x \rangle$ is a trace and $(tr_c, \{x\})$ is also a failure for some $x \in L'$. Then, by the refinement property of failures inclusion, both these must hold in the abstract system too and hence would contradict the assumption of limited security for the abstract system. So the limited security property holds through refinement. This does not prevent low level actions within $L \setminus L'$ from being refined in an insecure way. For example, in

the present case manipulating the frequency, routing or content of encrypted messages could convey information to the intruder.

This approach will not be suitable for all systems because the source of nondeterminism to be discounted may be bound up with other aspects which cannot be treated so lightly. For instance, in Section 7.3 the first attempt at specifying $C1NET1$ suffered from nondeterminism in the allowed ordering of messages output to low-level users. However, it would not be wise to discount the *receive* action when considering nondeterminism since the actual messages output might depend on high-level actions resulting in nondeterminism in the obscured system which should be dealt with.

For systems where a suitable separation presents itself it would be possible to perform an incremental proof of security at different levels of refinement. For example, suppose at a later stage or for a particular implementation of a network it is decided that it is *not* acceptable for even enciphered messages to appear as sent within the system (since, for example, increased traffic to a particular destination may give much away). It would be possible to add further constraints which mask the effect, such as multiple broadcasting and added noise. If this could be done as a refinement of the original then the top level security checks need not be repeated, but an additional proof of a suitably-constructed determinism property for encrypted outputs could also be considered.

7.6.4 Further security considerations for link encryption

One of the assumptions implicit in the link encryption model is that, whilst links may be tapped, information held on all nodes is secure. However, if it is further suspected that any message which appears in plaintext on an intermediate node could be misused, then the specification should reflect this additional possible source of leaked information.

Such a leak may not be the result of intruder action, but is an unfortunate consequence of too much information being available to intermediate nodes. However, it is certainly something which may need to be legislated against

on a system-wide basis. It is reasonable to require that messages should only be read by the recipients for whom they are intended. It may also be the case that certain nodes are not secure and that an intruder could gain access and thus see in plaintext all messages which were routed through that node. A leak will therefore be modelled as additional output to a node which occurs when communications for other destinations appear in that node's store. The security conditions should check whether the determinism property holds when the leak action is added and regarded as a low-level activity (allowing for encrypted messages as above).

$$\textbf{for } a \in \mathit{Node} \textbf{ action } \mathit{leak}_a \textbf{ out } t! : \mathit{TCOM} \ :-$$
$$: \left[\ t! \text{ in } \mathit{store}(a) \wedge \mathit{dest}\ t! \neq a \ \right]$$

where for any $t : \mathit{TCOM}$:

$$\mathit{dest}\ t \mathrel{\widehat{=}} \mathit{first}\ (\mathit{second}\ t)$$

which is, the intended recipient node obtained from the envelope of the communication. The constraint that $\mathit{dest}\ t \neq a$ ensures that only communications for other nodes are considered as leaks. The requirement that $t! \in \mathit{TCOM}$ means that only unencrypted communications are regarded as leaks.

If security is assessed for *C2NET2* with *leak* added it is clear that since communications will appear in plaintext on intermediate nodes the deterministic security conditions can certainly not be met. This is to be expected since link encryption does not protect information on intermediate nodes.

7.7 End-system level encryption

A message can be protected throughout the length of its journey from sender node to receiver node in several different ways. Here we describe a method of end-system level encryption. In terms of protocol layers, the encryption is performed much higher up (for example, at the transport layer) than the physical level and does not have to be reversed on each intermediate node.

The message is encrypted using the public key of the intended eventual recipient node. As before, this only requires one public key per node but it must be performed by the software for the appropriate layer rather than embedded in hardware. Intermediate nodes will now only see encrypted messages. Some header information (notably that concerning routing) will be transmitted in clear, but the information added at higher protocol layers will be protected by encryption. In this view of system security, although messages within envelopes are encrypted, the overall communication will not again be encrypted for forwarding over specific links. Hence the original state schema, $NS2$ defined on page 156 can be used. This defined stores and links as simply containing tagged envelopes. The system itself is defined in exactly the same way as $NET2$ in Figure 7.3 except that messages are encrypted on sending and decrypted upon receipt. We also add the actions representing the tapping of links and the output of information from the stores of intermediate nodes. For clarity, the whole is written out again as action system $C3NET2$ in Figure 7.8. Once again, any action of the legitimate network users will influence the intruder's view, but again, we wish to discount the effect of encrypted messages. In the case of link encryption, encrypted messages were set aside by allowing them to be ignored when checking the determinism criteria. The same could be done here, but the structure of the communications reveals that header information is also conveyed, and this appears in plaintext whether or not the message itself is encrypted. Excluding actions from consideration in the security conditions is a rather "blunt instrument" and it would be preferable to have a means of setting out exactly what our assumptions are about the way an intruder might gain information. So for end system encryption we try a different approach which expresses the output to the intruder by means of a function. The function describes how information is derived from the intercepted communications, allowing a complete and self-contained description of this to be set out. The specification can if desired be investigated with different definitions of information function to see which representations are consistent with maintaining system security and which would indicate a problem.

$C3NET2 \triangleq$

$$
\left(\begin{array}{l}
\textbf{var } NS2 \\
\textbf{initially } \begin{array}{l} store, \\ link \end{array} : \left[\begin{array}{l} (\forall\, a : Node \bullet Store(a) = [\![]\!]) \wedge \\ (\forall (a,b) : net \bullet link(a,b) = [\![]\!]) \end{array}\right] \\
\textbf{for } s \in Node \textbf{ action } send_s \textbf{ in } r? : Node;\ m? : Mess\ :- \\
\qquad store\ : \left[\begin{array}{l} \exists\, i : Tag \bullet (s, r?) \in route(i)^* \wedge \\ \qquad store(s) = store_0(s) + \\ \qquad\qquad [\![(i, (r?, enc(pk_{r?}, m?)))]\!] \end{array}\right] \\
\textbf{for } r \in Node \textbf{ action } receive_r \textbf{ out } m! : Mess\ :- \\
\qquad store\ : \left[\begin{array}{l} \exists\, i : Tag \bullet (i, (r, enc(pk_r, m!))) \text{ in } store_0(r) \wedge \\ \qquad store(r) = store_0(r) - \\ \qquad\qquad [\![(i, (r, enc(pk_r, m!)))]\!] \end{array}\right] \\
\textbf{internal } forward\ :- \\
\qquad \begin{array}{l} store, \\ link \end{array} : \left[\begin{array}{l} \exists\, a, b, r : Node;\ i : Tag;\ m : Mess \bullet r \neq a \wedge \\ \qquad (a,b) \in net \wedge (i, (r, m)) \text{ in } store_0(a) \wedge \\ \qquad (a,b) \in route(i) \wedge (b, r) \in route(i)^* \wedge \\ \qquad store(a) = store_0(a) - [\![(i, (r, m))]\!] \wedge \\ \qquad link(a,b) = link_0(a,b) + [\![(i, (r, m))]\!] \end{array}\right] \\
\textbf{internal } relay\ :- \\
\qquad \begin{array}{l} store, \\ link \end{array} : \left[\begin{array}{l} \exists\, a, b, r : Node;\ i : Tag;\ m : Mess \bullet \\ \qquad (a,b) \in net \wedge \\ \qquad tcom(i, (r, m)) \text{ in } link_0(a,b) \wedge \\ \qquad link(a,b) = link_0(a,b) - \\ \qquad\qquad [\![(i, (r, m))]\!] \wedge \\ \qquad store(b) = store_0(b) + [\![(i, (r, m))]\!] \end{array}\right] \\
\textbf{action } tap \textbf{ out } ti! : INFO\ :- \\
\qquad : \left[\begin{array}{l} \exists\, a, b : Node;\ c : TCOM \bullet \\ \qquad c \text{ in } link(a,b) \wedge ti! = info\ (tcom\ c) \end{array}\right] \\
\textbf{for } a \in Node \textbf{ action } leak_a \textbf{ out } li! : INFO\ :- \\
\qquad : \left[\begin{array}{l} \exists\, t : TCOM \bullet \\ \qquad t \text{ in } store(a) \wedge dest\ t \neq a \wedge \\ \qquad li! = info(tcom\ t) \end{array}\right]
\end{array}\right)
$$

Figure 7.8 Specification for end system encryption

The type *INFO* is introduced to allow information to be either a message (which, as defined above, can be text, a key, or an encrypted message) or is said to be *nil*.

$$INFO ::= mi(\!|Mess|\!) \mid nil$$

The function gives the information derivable by the intruder from a communication, and its type is:

$$info : COMM \rightarrow INFO$$

With this, the intruder's action *tap* can be described as before, but with output modified by the *info* function. A full definition must be given for *info* reflecting the required security policy. For example, if only the unencrypted message part of a communication is considered as being able to convey any information then for any $k : KEY$; $c : TCOM$; $i : Tag$; $r : Node$; $m : Mess$ we can define:

$$info\ (ecom(k, c)) = nil$$

$$info\ (tcom(i, (r, m))) = \begin{cases} nil & \text{if } m \in \text{ran } enc \\ mi(m) & \text{otherwise} \end{cases}$$

There is still a barrier to the complete determinism of the intruder's view, and that is the fact that an output of *nil* still constitutes information derived from network activity. However, this can be dealt with as before and, after noting the possible consequences of covert channels, the deterministic security condition can be limited to check only those outputs where the information is not *nil*. Using this approach, the envisaged interaction of the intruder is defined using the *info* function and the only exception which ever needs to be made in the security conditions is for a *nil* output value. A similar approach can be taken with the *leak* action.

The type of the output from $C3NET2_{leak}$ is a tagged envelope, which is also the type of objects in the store of each node. This allows messages to be encrypted or in plaintext, but always shows the destination in the header as a plaintext item. In practice it would generally be necessary for some

destination and/or routing information to appear in this way when a message is in a node's store since the node must direct the message on its future path. It is possible to devise a system which could operate without this, for example where a fully encrypted communication is broadcast to every node and each node attempts to decrypt it with only the intended recipient succeeding. However, this is not a practical option for most networks. For this analysis the leaks can be represented as independent action systems in this way. However, if a more complex representation of derived information were used (such as considering the accumulation of intercepted communications) then it would be necessary to consider the effect of combined sources of information.

To investigate the security of the system the effects of both the *tap* action and the *leak* actions must be considered. So, in the same way as before, the result of obscuring $\{send, receive\}$ in $C3NET2$ must be checked. The property to check is that:

$$obs_{\{send,receive\}}(C3NET2)$$

is deterministic within $\{tap.i \mid i \neq nil\} \cup \{leak.i \mid i \neq nil\}$. The guard of $tap.i$ for a specific value i is:

$$(\exists\, a, b : Node;\ c : COMM \bullet c \text{ in } link(a, b) \wedge i = info\ c)$$

For the limited security condition applied here, only the guard of actions with $i \neq nil$ need be considered. This, given the definition of *info*, is equivalent to:

$$(\exists\, t : Tag;\ a, b, r : Node;\ m : Mess \bullet \\ tcom(t, (r, m)) \text{ in } link(a, b) \wedge m \notin \text{ran } enc \wedge i = mi(m))$$

By construction, the system never reaches any state satisfying this (nor any state from which internal actions could do so) because every message sent into the system is encrypted. A similar argument holds for *leak* since messages remain encrypted in the store of any node. This indicates that the system is secure within the stated limits.

This example emphasises the point that any verification of a security property is only as accurate as the property itself. For, although the network with end-system encryption appears secure in the above analysis, the situation is changed dramatically if the information function is defined differently. For example, if unencrypted header information is deemed to be confidential then the type *INFO* could be extended to include information about a recipient node and the *info* function could include this type of output also. In this case, there is a direct influence from any *send* action on the information output by *tap* and *leak*. So with this change in the view of system security, *C3NET2* would not be secure.

This shows that the analysis and understanding of security requirements is crucial to the overall specification, since the interpretation of what constitutes acceptable information flow will obviously govern the outcome of the security validation. The network with end-system encryption as currently specified does not protect or disguise destination information. As discussed above, some form of multiple broadcasting could be used to mask the true recipient. Link encryption in addition to end-system encryption gives protection on the link, but the header information would still appear in clear on all intermediate links.

7.8 End-to-end encryption

Another way to protect a message throughout the whole length of its journey from sender to receiver is to use end-to-end encryption. The sending application performs the encryption (at the highest level of OSI protocol layer) using the public key of the recipient. The encrypted message is passed down the layers of the protocol with additional routing and other information added at each level. Thus the node (which may host many applications) does not have to participate in the encryption process. The store of each node again receives tagged, encrypted messages. This method provides a security tunnel not simply between nodes, but directly between applications and users.

The encryption process is now represented at the application level. Here,

we simplify by introducing user activity which is seen merely as generating and receiving messages for the purpose of demonstrating the interface with the network. There may of course be many users running different applications, but at this stage we introduce no more than the idea of a user able to encrypt and decrypt messages. Even with this modest addition changes must be made to the structure of the specification. Messages are now directed at a user on a node rather than just a node, so envelopes need to carry this information. The set:

$$[USER]$$

represents the users of the system, and for each $u : USER$ a public key, pk_u, and the corresponding secret key, sk_u, are needed. An envelope is now represented:

$$ENV \mathrel{\widehat=} USER \times Node \times Mess$$

To avoid the problem of messages being sent to users at the wrong node, each request to send a message will be checked for the correct address. The function:

$$users : Node \rightarrow \mathbb{P}\ USER$$

is introduced to represent the sets of users found at each node. A message m may be encrypted with a public key:

$$enc(pk_u, m)$$

or decrypted with a secret key:

$$dec(sk_u, m)$$

The decryption process will only yield the original plaintext if m was produced by encryption with pk_u.

The action system $USER_u$ of Figure 7.9 gives the behaviour for user u. The variable *flag* is introduced to indicate when the system is ready to accept input. The emphasis of system security has now changed, since each

$USER_u \mathrel{\widehat{=}}$

$$
\left(\begin{array}{l}
\textbf{var } outcom_u : ENV;\ usrstore_u : \text{bag } Mess;\ flag_u : \{true, false\} \\
\\
\textbf{initially } \begin{array}{l} outcom_u, \\ usrstore_u, \\ flag_u \end{array} : \left[\begin{array}{l} flag_u = true \wedge \\ usrstore_u = [\!] \end{array}\right] \\
\\
\textbf{action } inmsg \textbf{ in } u? : USER;\ r? : Node;\ m? : Mess\ :- \\
\quad \begin{array}{l} outcom_u, \\ flag_u \end{array} : \left[\begin{array}{l} (flag_u)_0 = true \wedge u? \in users\ r? \wedge \\ outcom_u = (u?, r?, enc(pk_{r?}, m?)) \wedge \\ flag_u = false \end{array}\right] \\
\\
\textbf{action } send \textbf{ out } e! : Env\ :- \\
\quad flag_u : \left[\begin{array}{l} (flag_u)_0 = false \wedge flag_u = true \wedge \\ e! = outcom_u \end{array}\right] \\
\\
\textbf{action } receive \textbf{ in } m? : Mess\ :- \\
\quad usrstore_u : \left[\begin{array}{l} ((m? \in \text{ran } text \vee m? \in \text{ran } key) \wedge \\ \quad usrstore_u = (usrstore_u)_0 + [\!|m?|\!]) \\ \vee \\ (m? \in \text{ran } enc \wedge \\ \quad usrstore_u = (usrstore_u)_0 + [\!|dec(sk_u, m?)|\!]) \end{array}\right] \\
\\
\textbf{action } outmsg \textbf{ out } m! : Mess\ :- \\
\quad usrstore_u : \left[\begin{array}{l} m! \text{ in } usrstore_u \wedge \\ usrstore_u = (usrstore_u)_0 - [\!|m!|\!] \end{array}\right]
\end{array}\right)
$$

Figure 7.9 User specification

user is made directly responsible for encryption and hence for protecting information. The *send* and *receive* actions of each user are the counterparts to the actions of those names in the network specification. The action *inmsg* takes as input the details of a message to be communicated in the network and encrypts it in an appropriate way for the recipient. The converse decryption operation is performed by the user receiving the message, with the plaintext version output to the user by *outmsg*. This view of the system begins to look at a wider picture, taking into account not only the nodes and the medium of the network, but also the interaction of the end users.

The network *C4NET2* is not required to provide any security mechanisms, so it can be specified in exactly the same way as the basic network *NET2* of Figure 7.3. Again, the addition of *tap* and *leak* functions models intruder action. To show the interaction with the user applications, the network specification can be refined to the next level (as in [20]) representing each node as a separate action system, $C4NODE_n$, shown in Figure 7.10. The links of the network together form the action system *C4MEDIA* as shown in Figure 7.11. With these definitions the next level of the overall system can be constructed:

$$C4NET3 = ((||\ n \in Node \bullet C4NODE_n) \ ||\ MEDIA) \setminus (\{forward_a \mid a \in Node\} \cup \{relay_b \mid b \in Node\})$$

and

$$C4NET2 \sqsubseteq C4NET3$$

with proof of this following the similar level of refinement in [20].

Parallel composition can again be used to represent the interaction of the users with the nodes of the system. First we define:

$$SYSUSERS = (||\ u \in USER \bullet USER_u)$$

Then, if:

$$C4NET2\ ||\ SYSUSERS$$

$C4NODE_n \triangleq$

$$
\left(\begin{array}{l}
\textbf{var } store_n : \text{bag}(Tag \times Env) \\
\textbf{initially } store : \left[\, store_n = [\!] \,\right] \\
\textbf{action } send \textbf{ in } (u?, r?, m?) : Env :- \\
\qquad store_n : \left[\begin{array}{l} \exists\, i : Tag \bullet (n, r?) \in route(i)^* \wedge \\ \qquad store_n = (store_n)_0 + [\!|(i, (u?, r?, m?))|\!] \end{array}\right] \\
\textbf{action } receive_u \textbf{ out } m! : Mess :- \\
\qquad store_n : \left[\begin{array}{l} \exists\, i : Tag \bullet (i, (u, r, m!)) \text{ in } (store_n)_0 \wedge \\ \qquad store_n = (store_n)_0 - [\!|(i, (u, r, m!))|\!] \end{array}\right] \\
\textbf{action } forward \textbf{ out } b! : Node;\ i! : Tag;\ (u!, r!, m!) : Env :- \\
\qquad store : \left[\begin{array}{l} (i!, (u!, r!, m!)) \text{ in } (store_n)_0 \wedge r! \neq a \wedge \\ (n, b!) \in route(i!) \wedge (b!, r!) \in route(i!)^* \wedge \\ store_n = (store_n)_0 - [\!|(i!, (u!, r!, m!))|\!] \end{array}\right] \\
\textbf{action } relay \textbf{ out } a? : Node;\ i? : Tag;\ (u?, r?, m?) : Env :- \\
\qquad store_n : \left[\begin{array}{l} (a?, n) \in net \Rightarrow \\ \qquad store_n = (store_n)_0 + [\!|(i?, (u?, r?, m?))|\!] \end{array}\right] \\
\textbf{action } leak \textbf{ out } t! : TCOM :- \\
\qquad\qquad : \left[\, t! \text{ in } store_n \wedge dest\ t! \neq n \,\right]
\end{array}\right)
$$

Figure 7.10 Specification of individual nodes for end-to-end encryption

$C4MEDIA \mathrel{\widehat=}$

$$
\left(\begin{array}{l}
\textbf{var } link : net \to \mathrm{bag}(Tag \times Env) \\
\textbf{initially } link : \left[\, \mathrm{ran}\, link = \{\!|\}\, \right] \\
\textbf{for } a \in Node \textbf{ action } forward_a \textbf{ in} \\
\qquad b? : Node;\ i? : Tag;\ (u?, r?, m?) : Env :- \\
\qquad link : \left[\begin{array}{l} (a, b?) \in net \Rightarrow \\ \qquad link(a, b?) = link_0(a, b?) + \{\!|(i!, (u!, r!, m!))|\!\} \end{array}\right] \\
\textbf{for } b \in Node \textbf{ action } relay_b \textbf{ out} \\
\qquad a! : Node;\ i! : Tag;\ (u!, r!, m!) : Env :- \\
\qquad link : \left[\begin{array}{l} (a!, b) \in net \wedge (i!, (u!, r!, m!)) \text{ in } link_0(a, b) \wedge \\ link(a, b) = link_0(a, b) - \{\!|(i?, (u?, r?, m?))|\!\} \end{array}\right] \\
\textbf{action } tap \textbf{ out } t! : TCOM :- \\
\qquad\qquad : \left[\, \exists\, a, b : Node \bullet t! \text{ in } link(a, b)\, \right]
\end{array}\right)
$$

Figure 7.11 Specification of medium for end-to-end encryption

is secure, so is:

$C4NET3 \parallel SYSUSERS$

Again, if *leak* and *tap* are the actions of an intruder to the system, the guards cannot be enabled when all the messages are encrypted, so $C4NET2 \parallel SYSUSERS$ with all other actions obscured is deterministic, and hence both $C4NET2 \parallel SYSUSERS$ and $C4NET3 \parallel SYSUSERS$ are secure with respect to the given analysis.

7.9 Further considerations

Each of the models presented above represents a very simplified view of security for network communication, with a single mechanism represented in each. This is useful from the point of view of trying out the deterministic security properties and showing how they can be applied, but for most real networks such a simplified model would be of little use. In this section we

consider some further issues which arise when considering secure communications in many practical networks.

Of the models above, the most satisfactory from the concerned user's point of view is probably the end-to-end encryption model since it provides the means of creating a virtual secure communications tunnel by setting up an encrypted pathway direct from user to user. There is no need to rely on what the network may or may not do and there is no need to worry about the specific links over which the message may travel. The approach does introduce other considerations: the confidentiality of an encrypted message is only as assured as the strength of the encryption algorithm and the secrecy of the keys. With each user requiring keys, key distribution and authenticity of keys and users becomes an issue.

Another problem is that each individual user may not always follow best practice for security or may not have the capability to do so. In some cases, a structure may be imposed by, for example, the manager of a local area network. This could be at the node level, or perhaps by routing communications through a firewall. The question also arises of how different security policies which may exist on the different nodes of a network can be accommodated within the overall plan of system security. In practice, a combination of measures will usually be in force: users may choose whether or not to encrypt messages; nodes may add layers of encryption; firewalls may enforce various constraints. Different levels of encryption (or none at all) may be modelled by combining these aspects of the previous specifications.

When the full generality and diversity of security requirements within a system is considered the question arises of what any network-wide policy may hope to provide. An individual node or user may decide to encrypt some messages for communication while others are sent in plaintext. If it were known at the system level what messages were supposed to be kept secret then it would be possible to check that none of these ever appeared in plaintext on links or in intermediate stores. However, it does not seem reasonable that a system-level view should be expected to know what each individual node requires to be protected. The best that the network can be expected to do is

to maintain the confidentiality of messages that have been encrypted (by not revealing keys) and to assume that any communication containing plaintext messages may be viewed by anyone without compromising security.

This view of security puts more emphasis on the nodes' ability to manage their own security requirements. Nodes with different security policies may be attached to the network. It is not just individual nodes which may have specific security requirements. Sub-networks, such as the LAN of a particular company, may operate a comprehensive policy for the member nodes. Once connected to the wider network, this policy cannot be expected to be maintained by outside links and nodes. So the individual node or sub-network must ultimately be responsible for identifying trustworthy nodes if they wish to communicate outside their own secure territory. They may reasonably expect some system-level help in making this decision, as is discussed below.

A sub-network with a specific policy may in part be represented as an instance of one of the networks specified above, or perhaps as a variant of one which is tailored for a required policy. For example, a sub-network of nodes may co-operate to maintain a multi-level security policy. However, the point or points at which connections are made to a wider network will lead beyond this safe environment to where the assumptions and guarantees of the local network no longer hold. The specification of a sub-network must be expanded to show the effect of these external connections.

There are many different aspects of network security but here we continue to concentrate on the area of confidentiality in a network of communicating nodes. From this viewpoint we can attempt to answer the question of what a node or sub-network can reasonably expect to be provided on a system-wide basis. Firstly it seems reasonable to require that encrypted messages are only ever successfully decrypted by their intended recipients. As became clear in the previous sections, there is no way in general to prevent any communication passing through many intermediate nodes, so the secrecy of the messages has to be maintained by securing the secret keys rather than restricting the communications. The preceding examples have used the approach of a public key system, but the same would be true for any

cryptographic system: secret keys are expected to be kept secret. So perhaps one requirement which could be expressed and checked for at the system level is that no secret keys are revealed to anyone other than the owner. This could be done as before using an information function in conjunction with the actions *leak* and *tap*.

Another issue which affects the confidentiality of a message is the correct identification of the party with whom the sender wishes to correspond. Suppose sender s wishes to correspond with recipient r. If intruder i tricks s into thinking that i is in fact r, then s may well reveal confidential material to i by mistake. In a system with public key cryptography this can happen if i can promote their public key as r's. The usual solution is a system of certificates whereby a trusted intermediary with a known, trusted public key "signs" r's public key with their own secret key. In practice, a chain of certificates may be required, starting from a trusted source, with each certificate in the chain vouching for the next element in the chain, and ending with r's key.

Whether or not a provision of this nature is seen as the responsibility of the network depends on the approach taken. Certainly, the various constituent parts of the network need to co-operate in such a scheme for it to work properly. In practice there have been various different approaches for the implementation of such a scheme. The original plan of certification for the secure mail application, Privacy Enhanced Mail (PEM), envisaged a strict hierarchy of certification authorities, leading back to one central authority which would be ultimately responsible for each complete chain. Such a system could maintain a very strict check on authenticity of keys (in fact, the original PEM vision was of a global system secure enough for legally-binding commercial transactions to be carried out) but the problems involved in setting up the ultimate certification authority have led to a less ambitious implementation. However, any individual network or sub-network could implement something similar, with authentication thus viewed as a global issue for the network.

The fallback position for PEM has been a graded system with root au-

thentication provided by the Internet Society. Authorities at lower levels are certified to specified degrees of trust with correspondingly different requirements on the stringency with which they are assessed. Users must decide which level of assurance they require and only accept keys which can provide a chain of certificates each meeting the required standard.

In contrast, another product, Pretty Good Privacy (PGP), designed to maintain the confidentiality of communications is much less rigid in its approach to certification. Each user chooses whom to trust and can store or accept a certificate without recourse to external authorities. This approach places the decision of whom to trust in the user's hands.

Each of these approaches has its advantages and disadvantages. The point of interest here is that what is expected to be enforced at a global level for a network is different in each case. There is a balance between what is provided by the user and what responsibility must be taken by the user.

Another consideration for confidentiality is the possibility that a communication received from an intruder or a corrupt node could contain a "Trojan horse" program which could operate to divulge confidential information. It is worth noting that such a program can operate only within the boundaries of the local security policy. However, any channels which have been left open by that policy can be exploited. This will include any of the routes of information flow which have been discounted as acceptable risks in the local analysis of security. For example, a suspect node s may not be able to subvert another node n so that messages are sent directly from n to s. However, clever manipulation of messages (even encrypted ones) sent to other trusted parties could be used to signal information when observed by s.

Again, this problem is difficult to address at any global level. Each node or sub- network must take on the responsibility for guarding against such attacks if the risk is significant.

7.10 Summary

This chapter has made a general investigation of the way in which security constraints may be used in an action system specification. It outlined several different ways of adding confidentiality requirements to a basic description of communication in a network. The approaches used represent very different ways of viewing system security. Initially, system-wide classifications were used and an overall multi-level security policy was specified. The later approach, with no security levels but with confidentiality maintained by encryption, is more suited to the realities of general networks. The use of action systems has allowed us to explore the effect of encryption at different levels and to incorporate the security requirements into a full functional specification. Since the aim here was to define and compare a number of different approaches, this chapter did not present the development of any one particular method in detail. The next chapter takes a specific example and shows how a security policy can be formulated and formally defined for that system.

Chapter 8

A distributed security kernel: case study 2

The previous chapter explored some of the general issues involved with using action systems for the development of secure systems and with the application of the deterministic security properties in particular. The current chapter takes a specific example and follows through the specification, statement and proof of security properties, and definition and proof of several levels of refinement for this single application. The problem area, though simplified from its original form, is a real application which has been researched and developed by members of Odyssey Research Associates Inc. (ORA). The case study was chosen as an example of a useful, sizeable system which seemed suited to an action system approach. The choice was not influenced by any attempt to find a system for which the deterministic security properties would be ideal, indeed, part of the purpose of the investigation is to see if the deterministic security properties do have a rôle to play in forming a security policy for a general purpose system.

8.1 Description of a security kernel

This case study involves the specification of a security kernel. This is basically a data base of items to which access may be restricted in various ways. The kernel serves a network of distributed hosts, receiving access requests and

checking to allow legitimate access only. The kernel can be refined so that it itself is distributed amongst the various hosts. Thus each host will each have its own local kernel component and these kernel components can interact to make decisions about non- local accesses. The combined effect of the kernel portions working together is the same as the top-level definition of a single, overall kernel.

The area for the case study is loosely based on descriptions of the kernel of a secure distributed operating system developed at ORA [54, 121]. The scope of the specification has been reduced and some aspects changed to allow a full specification and several layers of refinement to be represented here, but the basic concept is the same. The analysis performed by the authors [54] describes the possible approaches considered for this project. It is interesting to find that although the authors note the usefulness of process algebras for describing such things as kernel to kernel communication, the intricacy of such specifications and the need to represent state and invariants on that state led to the eventual choice of Larch [55] for specifications. The authors are well aware of the variety of information flow properties available, referring to the "cottage industry" in the development of such properties. Their decision in the face of this is to choose Bell and LaPadula with a "separate and somewhat *ad hoc* argument that information flows through other channels ... are not a serious threat" [54]page 287. The problems caused by the Refinement Paradox are also noted. No solution can be provided, but the authors suggest that any security analysis must somehow take account of the refinement level currently of interest.

The kernel restricts access to a persistent object database (other aspects such as authentication are not addressed here). Each host must operate within a specified security range, so all information accessed by that host must be within its range. A user currently active on a host can, via an application, request access to a database object. The user and the application must both be registered with the kernel and must be operating within their allowed security ranges. The access request (viewed here as a request for permission to view information stored in a database object) is processed by

the kernel and a decision is made based on the security information stored by the kernel.

8.2 Abstract specification of the kernel

The state of the system is a very important aspect to model for this system and, even for the simplified system described here, can become quite complex. Z [125] is used here to specify the state of the system. As mentioned previously, this is just one possible approach that can be used to specify the state of an action system. An overview of Z syntax is included in Appendix C.

8.2.1 The state of the kernel

The given sets for the specification are:

$$[HNAME, CLASS, APPID, OBJECT, USER]$$

Hosts, applications and users of the system are uniquely identified by elements of *HNAME*, *APPID* and *USER* respectively. *OBJECT* is the set of objects in the database administered by the security kernel. *CLASS* is the set of security classifications. For simplicity it is assumed that *CLASS* is linearly ordered and that it has least element $\bot$. The set of (possibly empty) security ranges is defined:

$$range\ CLASS == \{c1, c2 : CLASS \bullet c1 \mathinner{..} c2\}$$

A further type is used to denote the access decisions made by the kernel. Since we are here interested in one sort of access (reading an object) the result of a request will simply be denoted by a "grant" message or a "denied" message:

$$Reply ::= grant\langle\!\langle OBJECT \rangle\!\rangle \mid denied\langle\!\langle OBJECT \rangle\!\rangle$$

In this simplified system the important aspects of a host are the users currently active on it and the applications available to them. This is specified by the schema, *Host*1 given in Figure 8.1. Identifiers used here at the top level

*Host*1

$users1 : \mathbb{P}\ USER$

$appIns1 : \mathbb{P}\ APPID$

Figure 8.1 The state of a host

*Kernel*1

$userclear1 : USER \rightarrow range\ CLASS$

$hrange1 : HNAME \rightarrow range\ CLASS$

$dbase1 : OBJECT \rightarrow CLASS$

$currentusers1 : HNAME \rightarrow (USER \nrightarrow CLASS)$

$regapps1 : HNAME \rightarrow (APPID \nrightarrow range\ CLASS)$

$kernelin1 : \text{bag}(HNAME \times APPID \times USER \times OBJECT)$

$kernelout1 : \text{bag}(HNAME \times APPID \times USER \times Reply)$

$\forall u : USER;\ h : HNAME \mid u \in \text{dom}(currentusers1\ h) \bullet$

$(currentusers1\ h)\ u \in userclear1\ u \wedge$ **1**

$(currentusers1\ h)\ u \in hrange1\ h$ **2**

$\forall a : APPID;\ h : HNAME \mid a \in \text{dom}(regapps1\ h) \bullet$

$(regapps1\ h)\ a \subseteq hrange1\ h$ **3**

Figure 8.2 The state of the kernel

of abstraction are identified by names ending in 1. The kernel is defined at this level as a single database. It has a number of components which model the security information needed for the system. The kernel is described by schema *Kernel*1 in Figure 8.2. Each user and each host must operate within a certain security range. The allowed ranges for each user and each host are recorded by the functions *userclear*1 and *hrange*1 respectively. The function *dbase*1 gives, for each object of the database, a corresponding security classification. Not all users or applications working on hosts will be currently registered with the kernel (and therefore will not be able to access database objects). The currently registered users on each host, together with the security level they are working at, are recorded by *currentusers*1. Similarly, the registered applications from each host are recorded along with the security

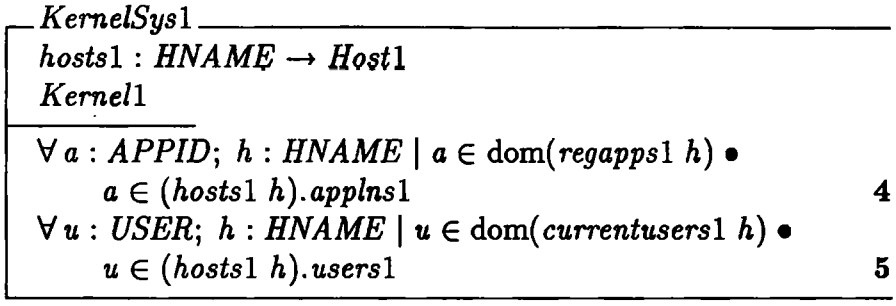

Figure 8.3 The state of the kernel

range within which they may operate. This is the purpose of *regapps*1. The component *kernelin*1 is a bag of access requests waiting to be dealt with by the kernel. A request to access an object is made by a user of a particular application working on a specific host. The final element *kernelout*1 holds the replies from the kernel waiting to be delivered to the correct recipient.

The predicates relating the schema components may be described as follows:

1. the clearance at which a registered user is currently operating is within the allowed range for that user.

2. the clearance at which a registered user is currently operating is within the allowed range for its host.

3. a registered application has a clearance range within that of its host.

As it stands, a user could be active on different hosts at the same time and with different current levels of security in each place. There is no problem with this for current purposes, but it might need to be ruled out in other situations.

Using the schemas representing individual hosts and the security kernel, the system is described by the schema *KernelSys*1 in Figure 8.3. The system consists of a collection of named hosts and a security kernel. The given constraints relate these in the following ways:

4. the applications registered with the kernel are all existing applications on the appropriate host.

5. a user registered with the kernel must indeed be a user on the stated host.

8.2.2 The top level action system

For the purposes of this specification we wish only to consider the activities of making access requests and receiving replies. Other matters such as the logging on and registration of users will be left aside. It will therefore be assumed that all state components except for *kernelin*1 and *kernelout*1 are fixed to some initial value obeying the state invariant and that they are unchanged throughout. When the system is refined, the corresponding initial settings are assumed for the concrete version. The action system representation of the system is given in Figure 8.4. It uses a functional decision procedure, *DECIDE*1, to allow or deny access requests. This is represented by the tree in Figure 8.5. Each branch of the tree represents a possible situation (within the guard of the action) and shows the corresponding command in each case. The situations described in the tree are mutually exclusive and so the choice of case is deterministic. This representation is intended to give a graphic picture of the basis on which a decision is made. It can be captured more conventionally as an alternative statement from which the weakest precondition may also be calculated.

Execution of *SecKer*1 proceeds as follows. Initially there are no input or output messages. A request to an object may be made from a host by an application invoked by a user. This action is always enabled and the input request is placed in *kernelin*1. The way in which a decision is reached by the kernel is represented at this level by an internal action, *decide*. There is no output to the user from this internal action. Whenever the *kernelin*1 bag is nonempty, a request may be selected, removed from *kernelin*1 and a decision placed in *kernelout*1 as specified by *DECIDE*1. Whenever *kernelout*1 is nonempty the output action, *response* is enabled. This can deliver any waiting reply, removing it from *kernelout*1.

$SecKer1 \mathrel{\widehat{=}}$

$$
\left(\begin{array}{l}
\textbf{var } KernelSys1 \\
\textbf{initially } kernelin1, kernelout1 := [\!], [\!] \\
\textbf{for } h \in HNAME \textbf{ action } invoke_h \textbf{ in} \\
\qquad\qquad a? : APPID;\ u? : USER;\ o? : OBJECT\ :- \\
\quad true \rightarrow kernelin1 := kernelin1 + [\!(h, a?, u?, o?)\!] \\
\textbf{internal } decide\ :- \\
\quad kernelin1 \neq [\!] \rightarrow (\textbf{var } (h, a, u, o) \in kernelin1 \bullet DECIDE1) \\
\textbf{for } h \in HNAME \textbf{ action } response_h \textbf{ out} \\
\qquad\qquad a! : APPID;\ u! : USER;\ r! : Reply\ :- \\
\quad (h, a!, u!, r!) \textbf{ in } kernelout1 \rightarrow \\
\qquad kernelout1 := kernelout1 - [\!(h, a!, u!, r!)\!]
\end{array}\right)
$$

Figure 8.4 The top level action system: *SecKer*1

8.3 The security of the kernel

The specification combines a fixed state of hosts, users, database objects and clearances with the actions for invoking objects and receiving responses. To analyse the security of the system solely in terms of events it would be necessary to model the complete history of the system, with values assigned to all the state components by visible actions and each such event classified according to the classification of the component it initialised. This is certainly possible, but it would involve a good deal of additional complexity. The additional specification needed would not be particularly useful in any other context and would make the overall effect of the specification less clear. Also, as remarked earlier, there are often additional properties to be proved apart from basic noninterference. These can often be stated most easily as invariants on the state. These considerations lead to the following analysis of security for *SecKer*1 in which both actions and certain security-sensitive state components are taken into account.

When considering the security of a system it is necessary to provide a definition of what constitutes security for that particular system. The step

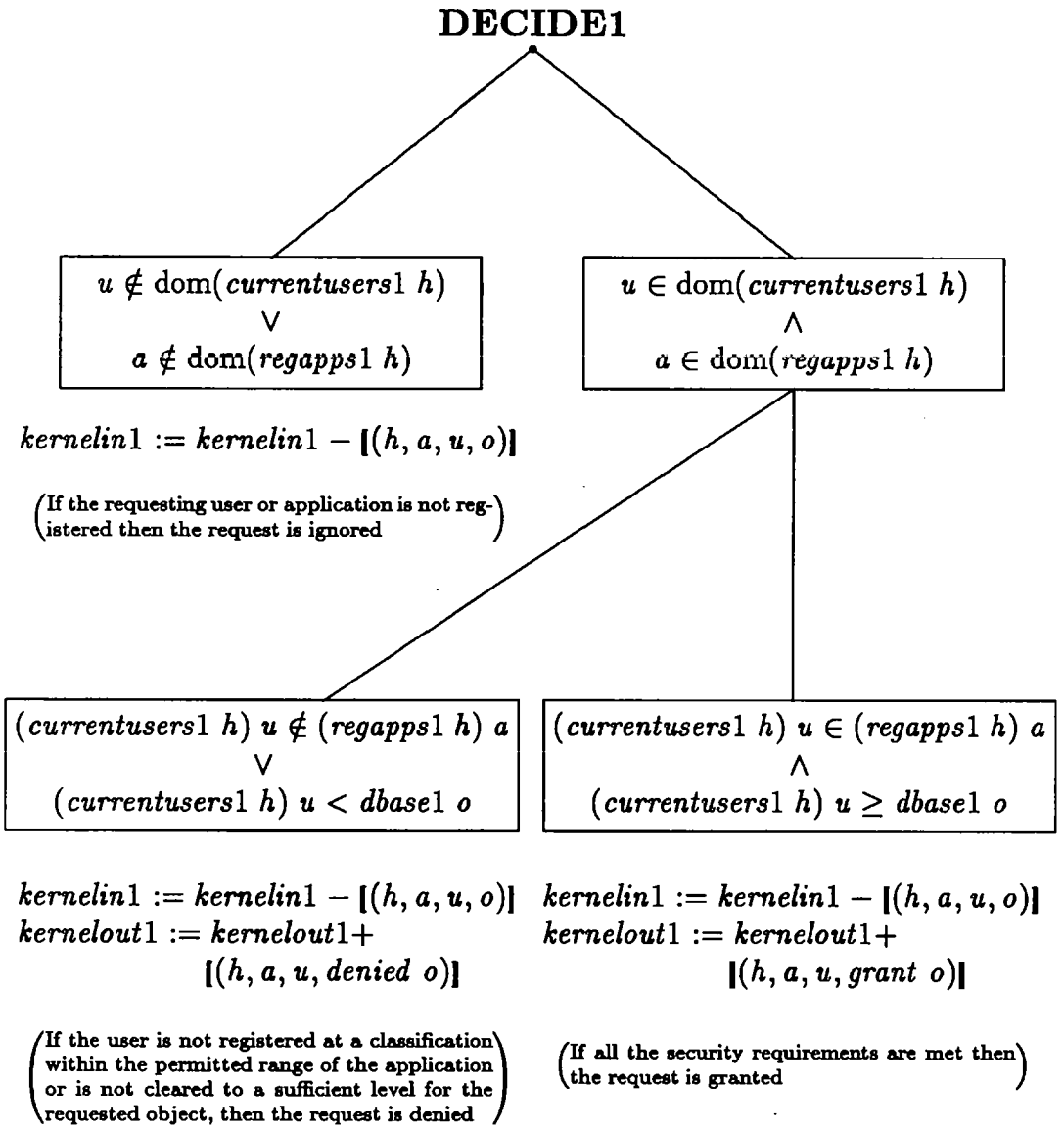

Figure 8.5 Tree describing *DECIDE*1

needed to create this is similar to performing a risk analysis for a safety-critical system to identify risks and to assess which are severe enough to try to prevent. Such an approach can help formulate a security policy against which a specification can be measured. Even for a system which could be viewed solely in terms of the deterministic security properties, it would still be necessary to categorise the actions as high-level or low-level, and this in itself represents a step of analysis. Here, the level of an action is identified with the level of the corresponding user (as specified by *currentusers*1 or $\perp$ if the user is not registered with the kernel). This allows the familiar noninterference requirement to be stated:

S1 no high level action should interfere with a low level action.

However, to protect the objects in the database, which have classifications in their own right, a more direct approach can be taken:

S2 no object with classification $\geq c$ is granted to a user with security level less than c.

Several other security requirements can be stated for the system:

S3 each user must work at a security level within the allowed range for its current host.

S4 a user cannot make use of an application unless their security level is within the allowed range for the application.

S5 each user must work at a level within their own allowed range.

The conditions **S1** ... **S5** represent the security policy selected for the system. **S3** and **S5** are incorporated directly into the specification as they are both part of the state invariant of *Kernel*1. **S2** is a familiar requirement similar to the "no read up" rule of Bell and LaPadula. **S1** is an information flow property which could make use of the deterministic security conditions. This is addressed in the following section.

The property **S2** will be ensured if, whenever the guard of *response* is enabled for output $(h, a, u, granted\ o)$, then the user's clearance must be at least as high as the classification of the object. That is:

$$((h, a, u, granted\ o) \in kernelout1) \Rightarrow \\ ((currentusers1\ h)\ u \geq (dbase1\ o))$$

That this property holds can be shown by induction over the actions of *SecKer*1 as shown in Appendix D. Property **S4** may be stated formally in a similar way, with $r \in Reply$:

$$((h, a, u, r) \in kernelout1) \Rightarrow ((currentusers1\ h)\ u \in (regapps1\ h)\ a)$$

For proof of this, see Appendix D.

8.4 Proving noninterference for the kernel

Suppose we wish to show **S1** with information flow defined as for the lazy deterministic security property. It would therefore be necessary to show that *SecKer*1 with high-level actions obscured is deterministic.

8.4.1 Obscuring high level actions in *SecKer*1

Taking an arbitrary security level c, the system with high-level actions (that is, ones with user's clearance greater than c) obscured is given in Figure 8.6. Some extra notation is required to do this. High-level actions are those which are genuine requests (or replies) from a user working at classification greater than c. Requests from unregistered users will be regarded as having level $\bot$, and so will always be low-level. This is reflected in the definitions of *creq* and *cresp* which give the security level of a request and a reply respectively. For any $h : HNAME$, $a : APPID$, $u : USER$, $o : OBJECT$ and $r : Reply$:

$$creq(h, a, u, o) = \begin{cases} (currentusers1\ h)\ u & \text{if } u \in \text{dom}(currentusers1\ h) \\ \bot & \text{otherwise} \end{cases}$$

$$cresp(h, a, u, r) = \begin{cases} (currentusers1\ h)\ u & \text{if } u \in \text{dom}(currentusers1\ h) \\ \bot & \text{otherwise} \end{cases}$$

$obs_H(SecKer1) \mathrel{\widehat{=}}$

$$
\left(\begin{array}{l}
\textbf{var } KernelSys1 \\
\textbf{initially } kernelin1, kernelout1 := [\!|\,|\!], [\!|\,|\!] \\
\textbf{for } h \in HNAME \textbf{ action } invoke_h \textbf{ in} \\
\qquad\qquad a? : APPID;\ u? : USER;\ o? : OBJECT\ :- \\
\qquad creq(h, a?, u?, o?) \leq c \rightarrow \\
\qquad\quad kernelin1 := kernelin1 + [\!|(h, a?, u?, o?)|\!] \\
\qquad \mid\ creq(h, a?, u?, o?) > c \rightarrow \\
\qquad\quad kernelin1 := kernelin1 + [\!|(h, a?, u?, o?)|\!] \\
\qquad \mid\ creq(h, a?, u?, o?) > c \rightarrow skip \\
\textbf{internal } decide\ :- \\
\quad kernelin1 \neq [\!|\,|\!] \rightarrow (\textbf{var } (h, a, u, o) \textbf{ in } kernelin1 \bullet DECIDE1) \\
\textbf{for } h \in HNAME \textbf{ action } response_h \textbf{ out} \\
\qquad\qquad a! : APPID;\ u! : USER;\ r! : Reply\ :- \\
\qquad (cresp(h, a!, u!, r!) \leq c) \wedge ((h, a!, u!, r!) \textbf{ in } kernelout1) \rightarrow \\
\qquad\quad kernelout1 := kernelout1 - [\!|(h, a?, u?, o?)|\!] \\
\qquad \mid\ (creq(h, a!, u!, r!) > c) \wedge ((h, a!, u!, r!) \textbf{ in } kernelout1) \rightarrow \\
\qquad\quad kernelout1 := kernelout1 - [\!|(h, a?, u?, o?)|\!] \\
\qquad \mid\ creq(h, a!, u!, r!) > c \rightarrow skip
\end{array}\right)
$$

Figure 8.6 *SecKer*1 with high level actions obscured

8.4.2 Nondeterminism

The lazy deterministic security property requires that the obscured version of *SecKer*1 be deterministic. Since high-level actions are always enabled, and low -level inputs are always enabled, this is equivalent (by Theorem 4) to showing that low-level outputs are enabled deterministically. However, it is immediately clear that this cannot be the case because any reply in the bag of outputs may be chosen by the system, and so the system can behave nondeterministically towards the low-level user. This is similar to the problem with bags noted in the previous chapter. There it was suggested that one (albeit unappealing) way to deal with the problem would be to have a less abstract top level specification. Here it would not be possible to keep the same level of abstraction and the same structure and simply change bags to sequences. The overall aim of the development is to produce a distributed secure kernel system. A top level which used sequences could not be refined to a lower level of distributed sequences since the order of output actions cannot be guaranteed. There are definitions of refinement, such as Jacob's interleaving refinement [65], which would allow this, but since nondeterminism can be introduced by such a refinement, it is not helpful in this context.

Another approach is to leave the proof of security until a later level, in this case, when the distribution has been accomplished. However, there are several problems with this. Firstly, the proof of determinism becomes more difficult the more detailed the specification becomes. To make the proofs more manageable it is necessary to attack the problem at as high a level as possible. Secondly, it may not always be possible to remove all nondeterminism. There are systems, particularly when dealing with networks, where genuine nondeterminism lurks. When sending messages over a network it may not be possible to say which will arrive first and, indeed, if either will actually arrive at all. Once again, this shows the difficulty of applying such stringent conditions to the development of general systems.

For present purposes we note that manipulation of order of output is one possible source of information flow from high to low which should be recorded

and bourne in mind during the development. However, it is still legitimate to ask whether, if the order of output *could* be fixed, would any information flow from high to low remain? So the lazy deterministic security property will be applied to $SecKer1$ with sequences rather than bags even though the version with bags is the one to be refined. Again, as with the route taken in the previous chapter, this is an expedient which makes continued development possible, but it is not an ideal solution.

8.4.3 An approach to proving nondeterminism

Even for a simple system of the size of $obs_H(SecKer1)$, a proof of determinism can be difficult to achieve. As discussed in Chapter 5, the property must be proved for all possible traces of the system and for all guards. A sufficient condition would certainly be that each action makes no internal choice, but this is not a necessary condition. As demonstrated by system $\mathcal{A}2$ in Example 18, a system can involve internal choices and yet still be deterministic with respect to events (this was referred to as event nondeterminism). This may well be the case with $obs_H(SecKer1)$ (when sequences are used). Internal choices are possible for both high-level input and output actions, yet the system may be event deterministic.

To avoid a head-on attempt at proving this property we take another approach which can make proof easier. Suppose a non-divergent system *Simple* can be found which has deterministic actions (in the sense that no action makes an internal choice) and which is refined by $obs_H(SecKer1)$. By Theorem 3, *Simple* is deterministic and, since $Simple \sqsubseteq obs_H(SecKer1)$, the obscured $SecKer1$ system must also be deterministic. The proof then becomes one of refinement. This is generally a more manageable problem, and it is one for which a number of simplified rules already exist.

The first step is to choose an appropriate deterministic system, *Simple*. The expectation is that high-level actions will not interfere with low-level ones, so *Simple* is constructed by allowing high-level actions to be defined as *skip*. Several other definitions help to make the definition of *Simple* as clear

as possible. Firstly, let:

$$REQUEST == HNAME \times APPID \times USER \times OBJECT$$
$$OUTPUT == HNAME \times APPID \times USER \times Reply$$

The $DECIDE1$ choice deals with a single request. Its result is either to ignore the request (if invalid) or to place a decision in $kernelout1$. The function f is defined to be of type:

$$f : REQUEST \rightarrow \text{bag } Reply$$

with, for $(h, a, u, o) : REQUEST$:

$$f\ (h, a, u, o) = \begin{cases} [\!] \\ [\!|(h, a, u, denied\ o)|\!] \\ [\!|(h, a, u, granted\ o)|\!] \end{cases} \quad \text{as specified by } DECIDE1$$

The action system *Simple* is defined in Figure 8.7.

8.4.4 Showing refinement of *Simple*

It is necessary to show that:

$$Simple \sqsubseteq obs_H(SecKer1)$$

The refinement relation RR is given by:

$$RR == (kernelin0 = kernelin1 \upharpoonright c) \wedge (kernelout0 = kernelout1 \upharpoonright c)$$

where $\upharpoonright$ restricts the queues to elements with $cresp \leq c$ and $crep \leq c$ respectively.

Both systems contain internal actions and, although a direct proof of refinement is possible, once again the easiest approach is to use a simplified rule. Proof for systems without internal actions can be validated relatively easily. So one way to proceed is to view (for the purposes of the proof) the internal action *decide* as a visible action in both systems and to show refinement between these systems. If this can be achieved the required result

Simple $\widehat{=}$

$$
\left(\begin{array}{l}
\textbf{var}\ \ \mathit{kernelin0} : \text{bag}\ \mathit{REQUEST};\\
\qquad \mathit{kernelout0} : \text{bag}\ \mathit{OUTPUT}\\
\textbf{initially}\ \mathit{kernelin0}, \mathit{kernelout0} := [\,], [\,]\\
\textbf{action}\ \mathit{invoke}\ \textbf{in}\ r? : \mathit{REQUEST}\ :-\\
\qquad\qquad \mathit{creq}\ r? \leq c \rightarrow\\
\qquad\qquad\qquad \mathit{kernelin0} := \mathit{kernelin0} + [r?]\\
\qquad\quad \mid\ \ \mathit{creq}\ r? > c \rightarrow \mathit{skip}\\
\textbf{internal}\ \mathit{decide}\ :-\\
\qquad \mathit{kernelin0} \neq [\,] \rightarrow\\
\qquad\qquad (\textbf{var}\ m\ \textbf{in}\ \mathit{kernelin0} \bullet\\
\qquad\qquad\qquad \mathit{kernelin0}, \mathit{kernelout0} :=\\
\qquad\qquad\qquad\qquad \mathit{kernelin0} - [m], \mathit{kernelout0} + (f\ m))\\
\textbf{action}\ \mathit{response}\ \textbf{out}\ r! : \mathit{OUTPUT}\ :-\\
\qquad\qquad (\mathit{cresp}\ r! \leq c) \wedge (r!\ \text{in}\ \mathit{kernelout0}) \rightarrow\\
\qquad\qquad\qquad \mathit{kernelout0} := \mathit{kernelout0} + [r!]\\
\qquad\quad \mid\ \ \mathit{cresp}\ r! > c \rightarrow \mathit{skip}
\end{array}\right)
$$

Figure 8.7 The deterministic action system: *Simple*

would then follow since for any action systems $\mathcal{A}$ and $\mathcal{B}$ and set of actions X:

$$\mathcal{A} \sqsubseteq \mathcal{B} \Rightarrow (\mathcal{A} \setminus X) \sqsubseteq (\mathcal{B} \setminus X)$$

There is a further complication because making *decide* visible reveals that this action in the concrete system would be enabled more often than in the abstract system, and so this would not conform to the action system refinement conditions. To remedy this, we could split the concrete *decide* internal action into two internal actions which are together equivalent to the single internal action. This is justified by the Internal Split Rule (Property 5 from Chapter 6). With this rule the system $obs_H(SecKer1)'$ can be defined as shown in Figure 8.8. By the Internal Split Rule this system is equivalent to $obs_H(SecKer1)$. If *decide* is made external (for the purpose of verification as described above) the concrete action system still has one internal event, *decidehigh*. However, the abstract system *Simple* has none so the simplified refinement rule given as Property 4 on page 139 may be used. The proof in Appendix D shows that:

$$Simple \sqsubseteq obs_H(SecKer1)'$$

and so *Simple* is refined by $obs_H(SecKer1)$ also. The system *Simple* with bags replaced by sequences is event deterministic since each action is deterministic, and so there can be no information flow from high to low levels except by manipulating the ordering of output messages.

8.5 First refinement of the kernel

The first refinement views the database objects as each residing on a particular host. Each host has its own "portion" of the security kernel which is responsible for taking security decisions involving objects, users and applications resident on that host. If a required object is not resident on the host from which the request originates then the host kernel must communicate with the kernel of the object's host. At this level we still represent the whole

$obs_H(SecKer1)' \mathrel{\widehat{=}}$

$$
\left(\begin{array}{l}
\textbf{var } KernelSys1 \\
\textbf{initially } kernelin1, kernelout1 := [\![]\!], [\![]\!] \\
\textbf{for } h \in HNAME \textbf{ action } invoke_h \textbf{ in} \\
\qquad\qquad a? : APPID;\ u? : USER;\ o? : OBJECT\ :- \\
\quad creq(h, a?, u?, o?) \leq c \rightarrow \\
\qquad kernelin1 := kernelin1 + [\![(h, a?, u?, o?)]\!] \\
\quad [\!] \ creq(h, a?, u?, o?) > c \rightarrow \\
\qquad kernelin1 := kernelin1 + [\![(h, a?, u?, o?)]\!] \\
\quad [\!] \ creq(h, a?, u?, o?) > c \rightarrow skip \\
\textbf{internal } decide\ :- \\
\quad (kernelin1 \upharpoonright c) \neq [\![]\!] \rightarrow \\
\qquad (\textbf{var}(h, a, u, o) \text{ in } (kernelin1 \upharpoonright c) \bullet DECIDE1) \\
\textbf{internal } decidehigh\ :- \\
\quad kernelin1 - (kernelin1 \upharpoonright c) \neq [\![]\!] \rightarrow \\
\qquad (\textbf{var}(h, a, u, o) \text{ in } (kernelin1 - (kernelin1 \upharpoonright c)) \bullet DECIDE1) \\
\textbf{for } h \in HNAME \textbf{ action } response_h \textbf{ out} \\
\qquad\qquad a! : APPID;\ u! : USER;\ r! : Reply\ :- \\
\quad (cresp(h, a!, u!, r!) \leq c) \wedge ((h, a!, u!, r!) \text{ in } kernelout1) \rightarrow \\
\qquad kernelout1 := kernelout1 - [\![(h, a?, u?, o?)]\!] \\
\quad [\!] \ (creq(h, a!, u!, r!) > c) \wedge ((h, a!, u!, r!) \text{ in } kernelout1) \rightarrow \\
\qquad kernelout1 := kernelout1 - [\![(h, a?, u?, o?)]\!] \\
\quad [\!] \ creq(h, a!, u!, r!) > c \rightarrow skip
\end{array}\right)
$$

Figure 8.8 The obscured system after applying the Internal Split Rule

system as a single action system, although the state is now defined in a way more suggestive of a distributed kernel.

At the previous level the bags *kernelin*1 and *kernelout*1 store the communications between the users and the kernel. Here, in addition to this the kernel on each host will have a bag of pending messages. This contains valid requests (that is, ones for registered users and applications) which may be for objects on this host or may need to be passed on to another host. It also contains replies which may be for delivery to a user on this host or again may need to be passed to another host for delivery. The pending queues will thus be used to store communications between kernels. So that both requests and replies can be held together we introduce the type *PMsg*:

$$\begin{aligned} PMsg \ ::= \quad & request\langle\!\langle OBJECT \times CLASS \rangle\!\rangle \\ | \quad & decision\langle\!\langle Reply \rangle\!\rangle \end{aligned}$$

The classification in a request represents the current security level of the requesting user. This is information which the user's host kernel will know, but if the requested object is on a different host the security level will need to be communicated with the request. The type of pending elements will be similar to those in *kernelin* and *kernelout*:

$$PENDING == HNAME \times APPID \times USER \times PMsg$$

It is convenient to define the type:

$$MT ::= req \mid dec$$

and the function:

$$\begin{array}{|l} mtype : PENDING \rightarrow MT \\ \hline \forall p : PENDING \bullet \\ \quad mtype\ p = req \Leftrightarrow fourth\ p \in \mathrm{ran}\ request \wedge \\ \quad mtype\ p = dec \Leftrightarrow fourth\ p \in \mathrm{ran}\ decision \end{array}$$

When a request is passed to another kernel there are various ways in which the routing can be handled. For simplicity, we assume that there is a directory of locations for all objects and that each host has a copy of this.

Kernel2

$$
\begin{array}{l}
userclear2 : USER \rightarrow range\ CLASS \\
hrange2 : range\ CLASS \\
dbase2 : OBJECT \rightarrow CLASS \\
currentusers2 : USER \nrightarrow CLASS \\
regapps2 : APPID \nrightarrow range\ CLASS \\
kernelin2 : \mathrm{bag}(APPID \times USER \times OBJECT) \\
kernelout2 : \mathrm{bag}(APPID \times USER \times Reply) \\
pending : \mathrm{bag}\ PENDING \\
\hline
\forall u : USER \mid u \in \mathrm{dom}\ currentusers2 \bullet \\
\quad (currentusers2\ u) \in (userclear2\ u) \wedge \quad \mathbf{1} \\
\quad (currentusers2\ u) \in hrange2 \quad \mathbf{2} \\
\forall a : APPID \mid a \in \mathrm{dom}\ regapps2 \bullet \\
\quad (regapps2\ a) \subseteq hrange2 \quad \mathbf{3} \\
\mathrm{ran}\ dbase2 \subseteq hrange2 \quad \mathbf{6}
\end{array}
$$

Figure 8.9 The state of the kernel at the second level of refinement

Communications can then be viewed at this level as a simple transfer to the correct host. The directory is specified as a global function:

$$location : OBJECT \rightarrow HNAME$$

The kernel for each host is described by the schema *Kernel2* in Figure 8.9. Each host's kernel needs the full information about users' allowed security ranges since any user may sign on to any host. Thus *userclear2* is simply a copy of *userclear1* replicated in full on each host. The allowed security range of operation for the host is given by *hrange2*. The portion of the database allotted to this host is denoted by *dbase2*. The next four state components of *Kernel2* are essentially the same as the corresponding components at the previous level, but restricted to a single host. The new element, *pending*, is also included. In the predicate part of the schema, Constraints **1**, **2** and **3** have the same purpose as their counterparts at the top level. Constraint **6** requires the database objects resident on the host to have security levels within the range allowed for the host. It is assumed that there exist hosts with ranges spanning all possible security levels, making it possible for the

Host2

$users2 : \mathbb{P}\ USER$
$applns2 : \mathbb{P}\ APPID$
$Kernel2$

$\mathrm{dom}\ regapps2 \subseteq \mathrm{dom}\ applns2$ **4**
$\mathrm{dom}\ currentusers2 \subseteq \mathrm{dom}\ users2$ **5**

Figure 8.10 The state of the kernel with distributed kernel

KernelSys2

$hosts2 : HNAME \rightarrow Host2$

Figure 8.11 The state of the system, *KernelSys2*

database to be distributed and the security conditions maintained.

Each host is now viewed as having current users, applications and a local kernel. This is defined as *Host2* in Figure 8.10. Conditions **4** and **5** in this schema correspond to the similar ones at the previous level. The system is now described by a simple function representing named hosts. This is given as *KernelSys2* in Figure 8.11. The global function *location* is related to this by the condition:

$$\mathrm{dom}((hosts2\ h).dbase2) = location^{-1}(\!|\{h\}|\!)$$

The action system *SecKer2* given in Figure 8.12 describes the system at this level. The behaviour of the kernel is still represented by internal actions, but it has been split up to show the different aspects of kernel activity. Instead of the single *DECIDE1* function the process is now represented with two steps, *DECIDE2* and *DECIDE3* pictured in Figures 8.17 and 8.18. *DECIDE2* defines the action taken by a local host when a request is made. It checks that this is a valid request - that is, the user and the application must be registered and the user must be working at a security level appropriate for that level. For valid requests which have been passed on to the host of the requested object, *DECIDE3* makes the final decision on whether access is to

$SecKer2 \mathrel{\widehat=}$

$$
\left(\begin{array}{l}
\textbf{var } KernelSys2 \\
\textbf{initially } (\forall h : HNAME \bullet (hosts2\ h).kernelin2, \\
\qquad (hosts2\ h).kernelout2, (hosts2\ h).pending := [\![]\!], [\![]\!], [\![]\!]) \\
\textbf{for } h \in HNAME \textbf{ action } invoke_h \textbf{ in} \\
\qquad\qquad a? : APPID;\ u? : USER;\ o? : OBJECT\ :- \\
\quad true \rightarrow (hosts2\ h).kernelin2 := \\
\qquad\qquad (hosts2\ h).kernelin2 + [\![(a?, u?, o?)]\!] \\
\textbf{internal } getrequest \\
\textbf{internal } transfer \\
\textbf{internal } decide \\
\textbf{internal } deliver \\
\textbf{for } h \in HNAME \textbf{ action } response_h \textbf{ out} \\
\qquad\qquad a! : APPID;\ u! : USER;\ r! : Reply\ :- \\
\quad (a!, u!, r!) \in (hosts2\ h).kernelout2 \rightarrow \\
\qquad (hosts2\ h).kernelout2 := \\
\qquad\qquad (hosts2\ h).kernelout2 - [\![(a!, u!, r!)]\!]
\end{array}\right)
$$

Figure 8.12 The second level action system: *SecKer2*

be permitted or not.

The internal actions supporting this activity are described below. It is useful first to define the function *destination* which maps a pending message to the host to which it is en route. For a request, this will be the host on which the requested object is located. For a reply, it is the host of the requesting user. So:

$$destination(h, a, u, request(o, c)) = location\ o$$
$$destination(h, a, u, decision\ r) = h$$

The internal actions are defined separately below. This structures the specification a little by dealing with the details of internal events first. The guards and commands have been separated out to emphasise each. The first internal action *getrequest* defined in Figure 8.13 deals with a request present in any of

getrequest

guard
$(\exists\, h : HNAME \bullet (hosts2\ h).kernelin2 \neq [\!])$

command
(**var** $h : HNAME;\ a : APPID;\ U : USER;\ o : OBJECT \mid$
$(a, u, o) \in (hosts2\ h).kernelin2 \bullet DECIDE2)$

Figure 8.13 The internal action *getrequest*

transfer

guard
$(\exists\, h : HNAME;\ p : PENDING \bullet$
$(p \in (hosts2\ h).pending) \wedge (destination\ p \neq h))$

command
(**var** $h : HNAME;\ p : PENDING \mid$
$(p \in (hosts2\ h).pending) \wedge (destination\ p \neq h) \bullet$
$(hosts2\ h).pending, (hosts2\ (destination\ p)).pending :=$
$(hosts2\ h).pending - [\!]p[\!],$
$(hosts2\ (destination\ p)).pending + [\!]p[\!])$

Figure 8.14 The internal action *transfer*

decide

guard
$(\exists\, h : HNAME;\ p : PENDING \bullet (p \in (hosts2\ h).pending) \wedge$
$(mtype\ p = req) \wedge (destination\ p = h))$

command
(**var** $h : HNAME;\ p : PENDING \mid (p \in (hosts2\ h).pending) \wedge$
$(mtype\ p = req) \wedge (destination\ p \neq h) \bullet DECIDE3)$

Figure 8.15 The internal action *decide*

deliver

guard
$(\exists\, h : HNAME;\ a : APPID;\ u : USER;\ r : Reply \bullet$
$\quad (h, a, u, (decision\ r)) \in (hosts2\ h).pending)$

command
$(\mathbf{var}\ h : HNAME;\ a : APPID;\ u : USER;\ r : Reply \mid$
$\quad (h, a, u, (decision\ r)) \in (hosts2\ h).pending) \bullet$
$\quad\quad (hosts2\ h).pending, (hosts2\ h).kernelout2 :=$
$\quad\quad\quad (hosts2\ h).pending - [\![(h, a, u, (decision\ r)]\!],$
$\quad\quad\quad (hosts2\ h).kernelout2 + [\![(a, u, r)]\!])$

Figure 8.16 The internal action *deliver*

the *kernel*2 queues. The request is removed from the appropriate *kernelin*2 and the command indicated by *DECIDE*2 is executed. If the request is valid it will be placed in the host's pending queue. A request in a pending queue may be for the host which owns the queue or it may need to be transferred to a different host in order for an access decision to be made. The pending queue may also contain replies and, again, they may need to be sent to another host for delivery. This is done by the internal action *transfer* of Figure 8.14. Any request which is in the pending queue for the host of its requested object may be dealt with directly by the host. The action *decide* of Figure 8.15 describes how the request is removed from the *pending* queue and a reply (as defined by *DECIDE*3) put back on the same queue. The *transfer* action already described is responsible for moving the reply to the correct host. The final internal action removes a reply from a pending queue when it has reached the correct host for the requesting user. This action, *deliver*, is defined in Figure 8.16. The reply is placed in the *kernelout*2 component for that host ready to be communicated to the correct user.

The relationship between the two levels is as follows. The original *kernelin*1 queue can be constructed by collecting together the *kernelin*2 queues for each host, plus all the messages which are still pending. Similarly, *kernelout*1 is the collection of all *kernelout*2 queues. The following definition is used:

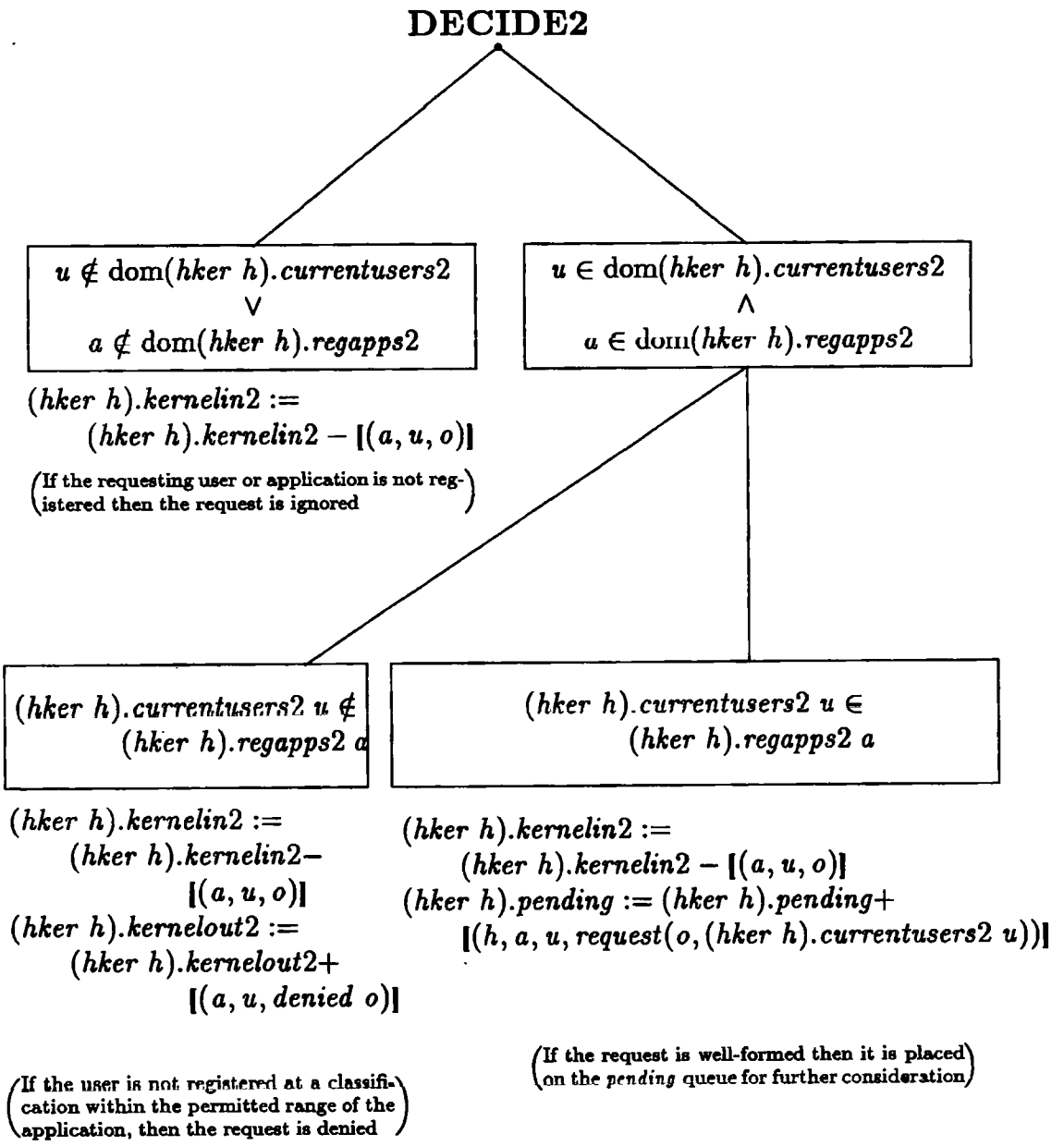

Figure 8.17 Tree describing *DECIDE2*

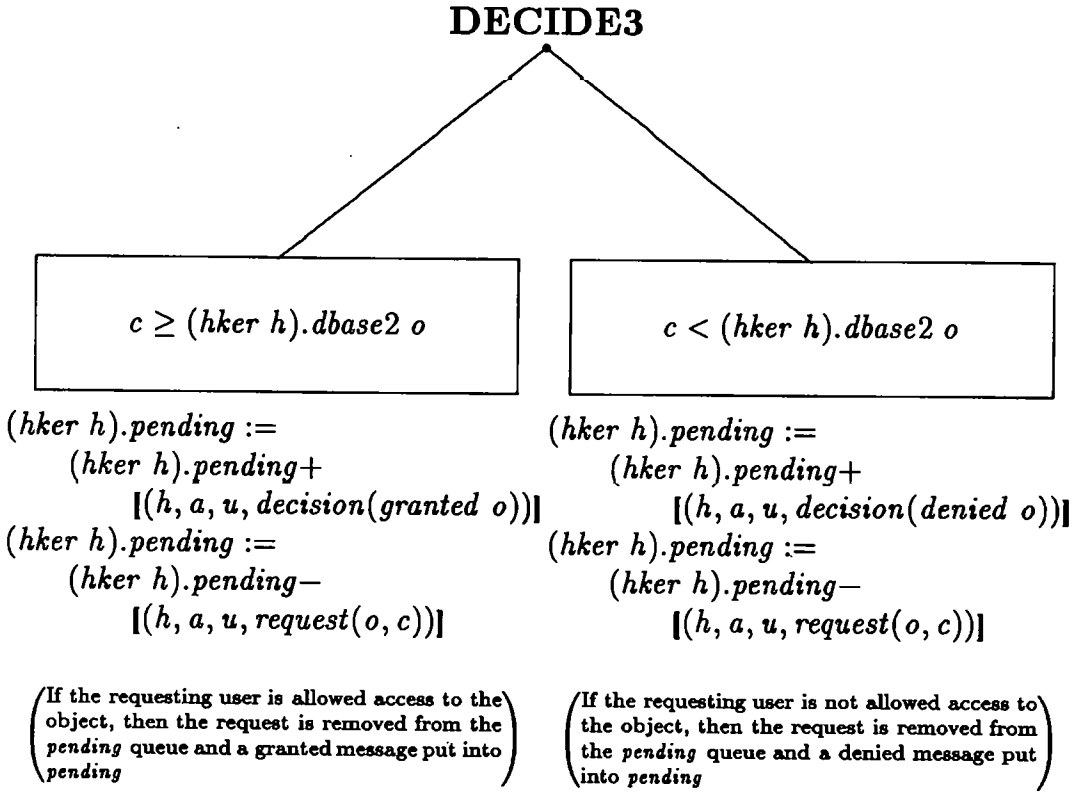

Figure 8.18 Tree describing *DECIDE3*

$$
\begin{array}{|l}
pendreqs : \mathrm{bag}\, PENDING \rightarrow \\
\qquad \mathrm{bag}(HNAME \times APPID \times USER \times OBJECT) \\
\hline
\forall b : \mathrm{bag}\, PENDING;\ h : HNAME;\ a : APPID;\ u : USER; \\
\qquad o : OBJECT;\ r : REPLY;\ c : CLASS \bullet \\
\quad (h, a, u, o) \in pendreqs\ p \Leftrightarrow \\
\qquad ((h, a, u, request(o, c)) \in b) \vee \\
\qquad (h, a, u, decision(granted\ o)) \in b \vee \\
\qquad (h, a, u, decision(denied\ o)) \in b \\
\qquad)
\end{array}
$$

If m is a message in $(host2\ h).kernelin2$ it can be mapped to the correct type for a $kernelin1$ message by including the information about its host, h, that is:

$$(a, u, o) \mapsto (h, a, u, o)$$

The function *origin* adds h to each message of the bag $(hosts2\ h).kernelin2$ in this way. The function *origout* performs a similar task for output messages. The retrieve relation $RR1$ can then be defined:

$$
\begin{array}{l}
RR1 \equiv \\
\qquad \begin{array}{lcl}
kernelin1 & = & \sum h \in HNAME \bullet origin\ (hosts2\ h).kernelin2) \\
 & & + \\
 & & \sum h \in HNAME \bullet penreqs\ (hosts2\ h).pending) \\
\wedge & & \\
kernelout1 & = & \sum h \in HNAME \bullet origout\ (hosts2\ h).kernelout2)
\end{array}
\end{array}
$$

Once again, both the concrete and the abstract systems have internal actions. The internal actions of the concrete system are intended to have the same overall effect as the single internal action of the abstract system, although when several messages are present there are various ways in which concrete internal actions can be interleaved. The external actions and initialisation are equivalent (via the retrieve relation) and the correspondence of the internal activities ensures that exactly the same traces and failures are available in both $SecKer1$ and $SecKer2$.

8.6 The distributed system

For this step, the system is represented as a collection of action systems, one for each host. At the previous level, the state was described as a function from host names to individual host states, so the work is already partly done. Now the events are divided in a similar way, with the overall system being described by a parallel composition of the individual host action systems. The state of a host is *Host2* defined at the previous level. We will refer to the state of host n as $Host2_n$ with all component names similarly subscripted with n. The action system for host n is given by $SecKer_n$ in Figure 8.19. Again, the internal actions $getrequest_n$, $decide_n$ and $deliver_n$ have been de fined separately in Figures 8.22, 8.23 and 8.24. These are basically the same actions as at the previous level but confined to a single host. The corresponding decision trees *DECIDE*4 and *DECIDE*5 are given in Figures 8.20 and 8.21. The main difference is with the action *transfer*. This was an internal action at the previous level, but here it is made visible to represent communication between the different parts of the kernel on different hosts. The *transfer* action will appear differently to a participant depending on the rôle they play. If a message is transferred from node a to node b then it is an output action for a and an input action for b. These differences are reflected in the way that $transfer_{(n,m)}$ is defined, with the two cases distinguished.

The overall system is formed by the parallel composition of all the individual host systems. Actions with like names are composed as described by the parallel composition operator defined in Section 4.4. These consist of the actions $transfer_{(n,m)}$ which will be joined in sender/receiver pairs. The communication channels described by the *transfer* actions are then hidden. The parallel decomposition achieved by this is given by:

$$SecKer3 = (\|\ n \in HNAME \bullet SecKer_n) \setminus \{m, n : HNAME \bullet transfer_{(m,n)}\}$$

Since *hosts*2 is a total function on *HNAME* we identify each $Host2_n$ with the corresponding $n \mapsto (hosts2\ n)$ from the previous level. So statements involving concrete variables will be equated with the same statement with

$SecKer_n \triangleq$

$$
\left(\begin{array}{l}
\textbf{var } Host2_n \\
\textbf{initially } kernelin2_n, kernelout2_n, pending := [\,], [\,], [\,] \\
\textbf{action } invoke_n \textbf{ in } a? : APPID;\ u? : USER;\ o? : OBJECT\ :- \\
\quad true \rightarrow kernelin2_n := kernelin2_n + [(a?, u?, o?)] \\
\textbf{internal } getrequest_n \\
\textbf{for } m \in HNAME \textbf{ action } transfer_{(n,m)} \textbf{ out } p! : PENDING\ :- \\
\quad m \neq n \wedge (p! \in pending_n) \wedge (destination\ p! = m) \rightarrow \\
\qquad\qquad pending_n := pending_n - [p!] \\
\textbf{for } m \in HNAME \textbf{ action } transfer_{(m,n)} \textbf{ in } p? : PENDING\ :- \\
\quad true \rightarrow pending_n := pending_n + [p?] \\
\textbf{internal } decide_n \\
\textbf{internal } deliver_n \\
\textbf{action } response_n \textbf{ out } a! : APPID;\ u! : USER;\ r! : Reply\ :- \\
\quad (a!, u!, r!) \in kernelout2_n \rightarrow \\
\qquad\qquad kernelout2_n := kernelout2_n - [(a!, u!, r!)]
\end{array}\right)
$$

Figure 8.19 The action system: $SecKer_n$ for host n

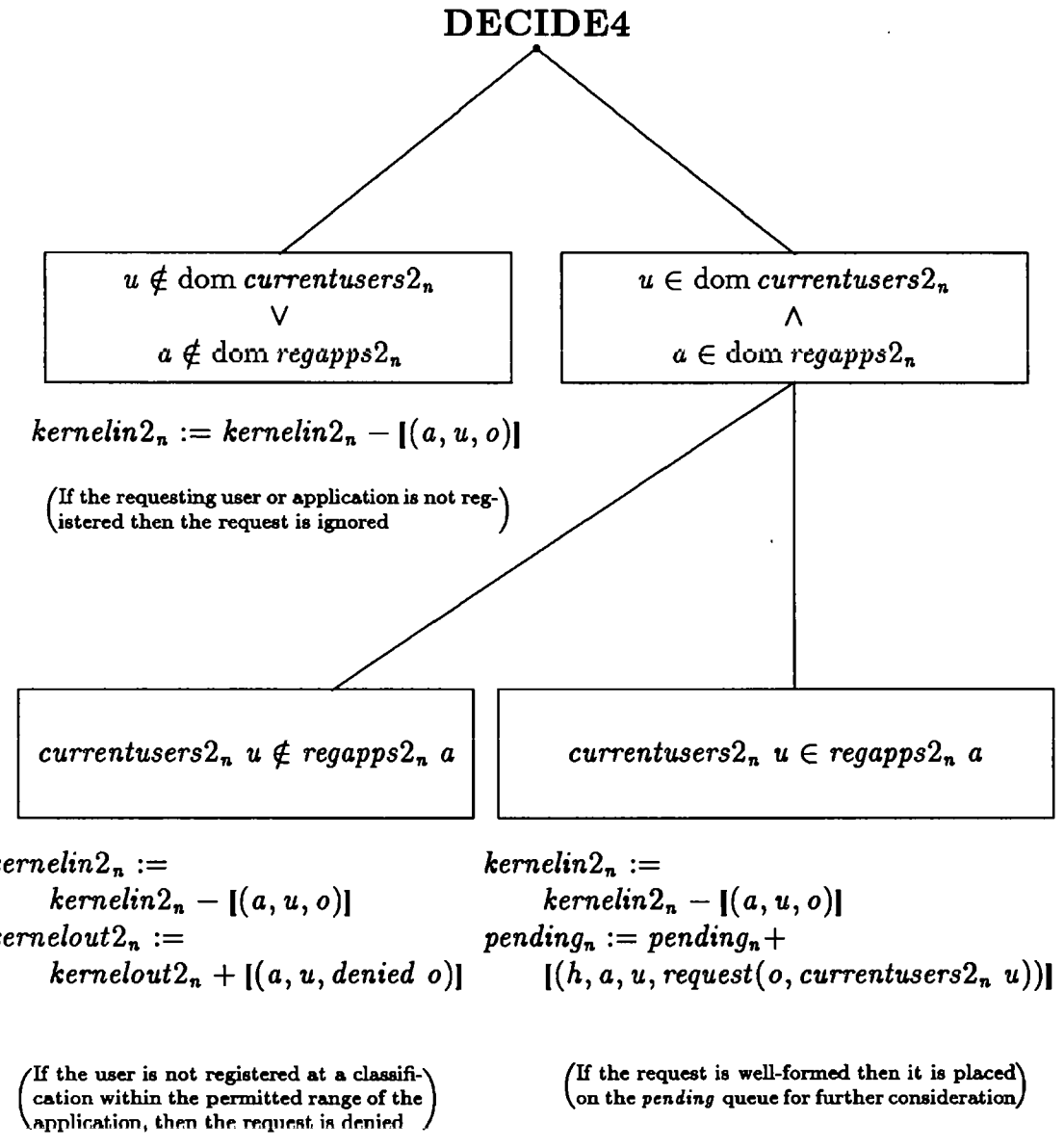

Figure 8.20 Tree describing *DECIDE4*

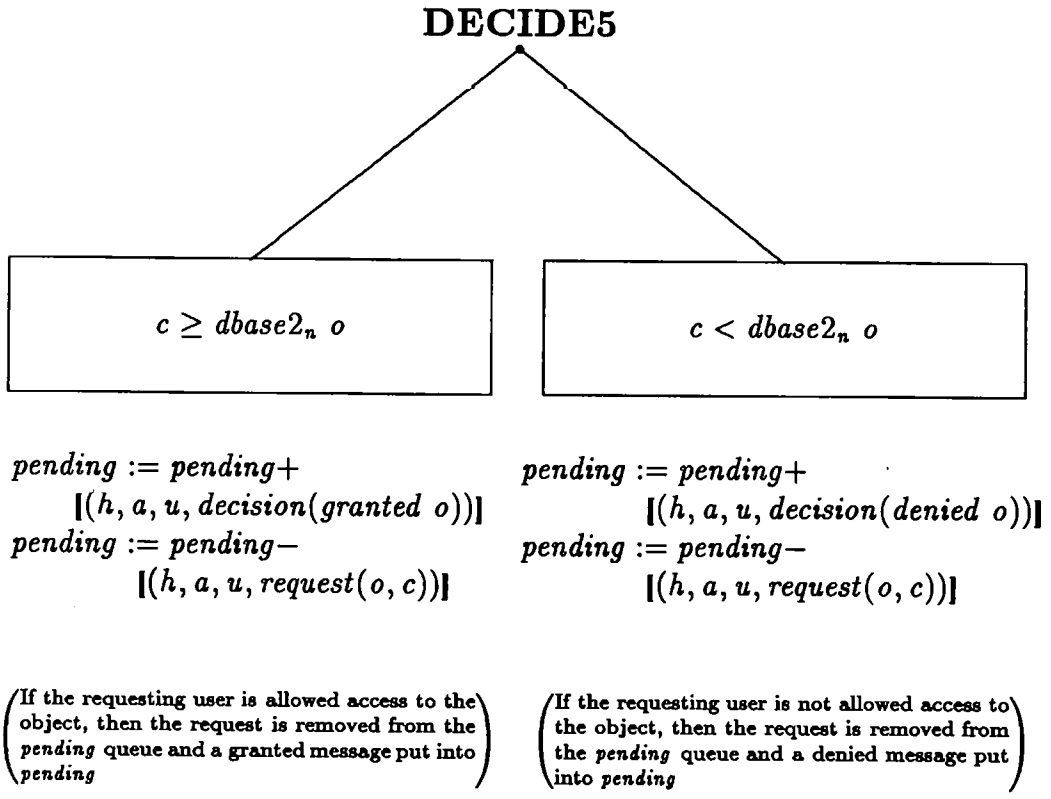

Figure 8.21 Tree describing *DECIDE5*

getrequest

guard
$kernelin2_n \neq [\!]$

command
(**var** $a : APPID;\ u : USER;\ o : OBJECT \mid$
$(a, u, o) \in kernelin2_n \bullet DECIDE4$)

Figure 8.22 The internal action $getrequest_n$

decide

guard
$(\exists\, p : PENDING \bullet (p \in pending_n) \wedge (mtype\ p = req) \wedge$
$(destination\ p = n))$

command
(**var** $p : PENDING \mid (p \in pending_n) \wedge (mtype\ p = req) \wedge$
$(destination\ p \neq h) \bullet DECIDE5$)

Figure 8.23 The internal action $decide_n$

deliver

guard
$(\exists\, a : APPID;\ u : USER;\ r : Reply \bullet (n, a, u, (decision\ r)) \in pending_n)$

command
(**var** $a : APPID;\ u : USER;\ r : Reply \mid$
$(n, a, u, (decision\ r)) \in pending_n) \bullet$
$pending_n, kernelout2_n := pending_n - [\!|(h, a, u, (decision\ r)|\!],$
$kernelout2_n + [\!|(a, u, r)|\!])$

Figure 8.24 The internal action $deliver_n$

corresponding abstract variables. For example:

$$kernelin2_n := kernelin2_n + [\![(a?, u?, o?)]\!]$$

is equivalent to:

$$(hosts2\ n).kernelin2 := (hosts2\ n).kernelin2 + [\![(a?, u?, o?)]\!]$$

Proof of this refinement is given in Appendix D.

8.7 Remarks

The development described here gives a top level specification and two levels of refinement for the security kernel. Security properties are stated and proved at the top level. Both the results about determinism and the trace properties are preserved by refinement. The refinements given here show how a single top level description can be decomposed into a distributed collection of action systems acting in parallel, with communication via synchronised value-passing. The development can be continued further towards various different implementations of the kernel system. For example, a kernel may operate in either "multi-level secure" mode or as "multiple single levels". For the latter, the communications at each security level are kept separate. Both of these approaches can be regarded as refinements of the same basic top level description.

Although the specification presented here is a simplified version of any real requirements for a security kernel, it outlines a general approach which could be expanded to include many other features of both the state and the interaction of events in the system.

The security analysis combines a number of requirements, with information flow viewed in terms of the lazy deterministic security property. With action systems, consideration of information flow does not have to be approached in an *ad hoc* fashion (which was one of the points noted with respect to the original Theta kernel specification [54]). The explicit description of event dependency means that the deterministic properties or other information flow conditions can be applied directly. This chapter introduces a

technique for facilitating proof of deterministic security properties. The approach reduces the verification of determinism to proof of refinement. This case study also highlights some of the problems of applying the deterministic security properties including the difficulty of dealing with "genuinely" nondeterministic features of a system. This is discussed further in the next chapter. Another difficulty is that of providing formal verification for all security properties and refinements. This again is discussed in Chapter 9.

The security of any system, however strictly verified, is still only as good as the security policy set out for it. In the case of the security kernel there are still other questions which might be asked. For example, the communication between host kernels is not made explicit until the second refinement level. Since the security analysis is done at a level where this communication is internal, it is in effect regarded as hidden from the outside users' view. So, if we have doubts about the security of the medium over which communication takes place it would be necessary to make that an explicit consideration and perform a security analysis which takes it into account at an appropriate level (perhaps using the techniques of the previous chapter).

Chapter 9

Conclusions and future work

The work presented here has brought together a number of different theories, techniques and notations to provide a pragmatic approach to the development of secure systems. It has drawn on the results of previous work in the field on its way to forming a unified approach. The work has involved consideration of a diverse range of notations including state- based, event-based, specification statements and wp calculus. The main aim has been to draw these threads together in a way which can incorporate some of the most important features of each. This chapter reviews the work of earlier chapters, presenting conclusions and comparison with similar work. Finally, we suggest future work arising from this research.

9.1 Discussion and comparison with other work

This work took as its starting point the various different approaches to achieving confidentiality outlined in Chapter 2. The information flow properties, though able to provide the basis for abstract security policies encompassing many possible sources of covert channels, are less intuitive and more difficult to apply than simpler methods such as access controls. The fact that such a variety of definitions exists makes it difficult for the practitioner to judge the merits of each and to decide which is appropriate to use. One of the problems of working with research into security is that, by the very nature of

the work, practitioners are often unable to provide information about what their requirements are and what methods they are employing. However, both available published reports (such as that of Boswell [16]) and informal communications suggest that the Bell and LaPadula model is still used and that Z is still a common notation. This contrasts with the concentration on information flow properties and event-based notations evident in much of the research into confidentiality over the past 15 years.

Given this apparent gap between theory and practice the approach of this thesis has been to use a formal notation, action systems, which can be used to express the various information flow properties and yet still retain some of the features of a popular, general- purpose specification language. This combines suitability for defining and proving information flow properties within a notation that provides equal support for defining the state and events of a system, together with the facility for both data and operation refinement. The case studies of the previous two chapters have shown how action systems achieve this and give an indication of the way in which such specifications can be structured and how levels of detail can be introduced by stepwise refinement.

The work of this thesis can be seen as both an investigation of the use of action systems for the development of secure systems in general and of the development of one specific approach (deterministic security) in particular. The former aspect is considered first.

9.1.1 The use of action systems

Action systems were chosen for this work both for their suitability in specifying state and events and for the relationship of the notation with CSP. This connection, formalised as shown in Chapter 4, provides a semantics for action systems in terms of CSP failures-divergences and infinite traces. This close association between the two notations means that various CSP operators can be defined directly for action systems. This in turn allows properties expressed in CSP to be defined for action systems, with the appropriateness of the translation confirmed formally with respect to the failures- divergence

semantics. This is useful since, as outlined in Chapter 2, many information flow properties have been expressed in terms of CSP.

The definition of action systems used here agrees with that of Butler [20] and differs slightly from the approach of Back and Kurki-Suoni [7]. The basic format of an action system is similar in the two views, but there are some important differences. The approach taken by Back does not distinguish between internal and external choice. This is a vital distinction to make when considering information flow properties, and hence makes Butler's approach much better suited to this type of security analysis. Another aspect to consider concerns the difference in refinement between the two approaches. Back defines parallel action systems which are connected by their shared state (rather than by similarly-labelled actions). This has the consequence that components of parallel compositions cannot be refined separately and independently, but have to be considered as a whole. The approach used here places greater emphasis on the events, with shared labels but no common state. Components of a parallel composition may be refined in isolation. This allows greater flexibility in development and a separation of concerns in refinement.

9.1.2 Comparison with CSP

Since the close link with CSP has been stated as an advantage it is perhaps necessary to ask what benefits the use of action systems brings. The main motivation, as stated above, is that state and events should be given equal treatment. The case studies of Chapters 7 and 8 show that action systems can be successfully employed for the specification and development of general-purpose systems, and the security constraints can be incorporated as part of an overall development. However, variables can also be used in CSP in an algebraic approach, with variables appearing as program parameters as with the process *STK* of Example 22. Just as the Back approach to action systems seemed to place greater emphasis on state, so the algebraic CSP approach is still mainly focused on representing event interaction. State and events must be refined separately in CSP, whereas the unified treatment provided

by action systems simulations justifies the refinement of an action system as a whole. Also, tools such as FDR [105] which appear very attractive for formal development of CSP specifications cannot deal with anything more than very limited state components. Thus, when using such a tool, the specification must be rendered in a format which takes this into account.

Another difference pointed out by Butler [20] is that in CSP the communication events for a single channel may occur in many places within a single process description. This can make reasoning about parallel composition more complicated than for action systems in which the composition of a pair of actions is all that is required.

Following on from the general points of difference between action systems and CSP, it is likely that the nature of an application may make it more suited to one or the other. For example, there has been much successful work carried out using CSP to model security protocols and to investigate their effectiveness. Roscoe [108] models the nodes and communications medium as CSP processes for a formal description of the Needham-Schroeder key exchange protocol. To investigate the security of the protocol, the assumed abilities of an intruder are also represented as a process operating in parallel with the rest of the system. The properties of interest are confidentiality (using deterministic security) and nondisruption (defined as a related property in which intruder behaviour must not affect the services available to legitimate users). Similar analysis is also carried out by Lowe [78, 79]. Schneider [119] also models both network and intruder behaviour with CSP, giving trace and process algebra definitions of confidentiality and other key properties for security protocols. This has led Schneider to develop a new library of CSP theory of particular use in the proof of security properties [120]. The key points for the suitability of this area of application to CSP are the need for detailed consideration of sequences of events required for the analysis of security protocols, together with the comparatively simple state requirements.

It is perfectly possible to model security protocols in action systems. This has been demonstrated by Butler's action system representation of the Needham-Schroeder protocol [19]. This shows the feasibility of such mod-

elling and gives a well-structured and readable presentation of the protocol. However, from the point of view of the security analysis of the protocol, action systems offer few advantages over CSP

Whilst analysis of security protocols fits well in a predominantly event-based notation, the case studies of Chapters 7 and 8 show that action systems provide an expressive notation for general system specification. Action systems can be viewed as extending existing state-based approaches by making clear the possible succession of events. This allows not only the information flow analysis described here, but also extends the notation to permit the description of distributed systems whose components work together in parallel. The state may be described and structured using existing notations (such as Z or specification statements) which have already been shown to be beneficial for this purpose.

It is often the case that security policies are formed from a combination of different security models. For example, the project reported by Boswell [16] uses Bell and LaPadula [11] for confidentiality, Clarke and Wilson [24] for integrity and two-person rule. Other policies may be even more complex and may be more naturally accommodated in a notation with explicit reference to state. For instance, a Chinese Wall policy setting out groups whose members may have conflicting interests can be described graphically in Z or action systems. These concepts go beyond the very specific information flow properties of confidentiality considered here, but are relevant to the applicability of action systems to a broader class of security developments.

Whilst comparing action systems with CSP it is important to consider not only the suitability of the specification but also the ease of development. The emergence of the FDR tool [105] has proved very useful for CSP developments and has been used to good effect for security-related work as described by Roscoe [108] and Lowe [78]. FDR is a CSP model-checker which automatically verifies refinements in the failures-divergence semantics. It can also be used for checking determinism, which means that the deterministic security properties can be automatically verified. FDR has been used to detect and discover a number of attacks on security protocols [78].

Further work continues to improve and extend FDR. Since the difficulty of obtaining the proofs of properties necessary in a formal development is well-known, the contribution of FDR is very welcome and makes use of CSP all the more attractive. Schneider [119] states that his CSP work on protocols has to some extent been motivated by the availability of model-checking tools. However, there are still some practical barriers which mean that successful use of FDR relies on careful preparation of the input specification. The main problem is the necessity to limit the state space. Even though the data sets required to specify a protocol may be small in number, the individual size of each may be large. For example, there may be many possible messages in the system and many possible keys. Roscoe [105] discusses ways for dealing with this by reducing data sets to small number of representative values, or by symbolic model checking. Research in this area continues, but current limitations mean that only carefully chosen and prepared specifications can be dealt with. Prospects for proof support for action systems are discussed below.

9.1.3 Comparison with state-based approaches

Although state-based approaches have frequently been used for security specifications (for example, see [122] for a survey of the use of Z) the difficulty of expressing interaction of events in such a notation has resulted in most research in this area being carried out in notations where such concerns are more naturally accommodated. However, there have been a number of initiatives to incorporate some of the lessons learned from information flow analysis into a state-based approach. This section outlines these attempts.

Work carried out by Collinson [25] based on the approach of the ICL Secure systems Group [5] uses Z to specify both the particular system under consideration and a more abstract state machine version which outlines the security constraints on the system. A correspondence must be made between the two to show how the transitions of the system match those of the state machine. Proof of the security conditions is then carried out for the interpreted specification. As demonstrated by Collinson, this approach can catch

errors that occur as the result of a succession of operations. The approach is for deterministic specifications only, and requires additional layers of specification and proof to interpret the system as a state machine. The complexity involved can hide important details. For example, although the specification is required to be functional, no check is made that this is so. The error of using a non-functional specification can vitiate the whole complex process designed to show proof of security.

In Appendix A of his thesis, Graham-Cumming [47] considers secure refinement for an abstract data type (ADT). ADT's are closely related to action systems, comprising state, initialisation and events. In effect, action systems provide a way of specifying the behaviour of an ADT. Graham-Cumming relates ADTs to CSP using the "garage map" which creates the process that first selects an initial state and then offers an external choice between events whose precondition is true. This approach gives a specific process, rather than the semantic characterisation of traces, failures and divergences used here. The security condition used is Graham-Cumming's noninterference property (described in Chapter 2). A Z specification viewed as an ADT can be translated to CSP using the garage map and the resulting process tested for noninterference. In contrast to the present approach, the security condition is not applied to the Z or the ADT but to the CSP translation. Also, no distinction is made between the precondition and guard of an action. This view differs from the usual interpretation of Z preconditions. It can also lead to significant divergence between the possible refinements of a Z specification and the possible refinements of the "equivalent" process.

Another approach to state-based specification of noninterference is described by Bevier and Young [13] whose system model resembles that of Goguen and Meseguer [43] with a set of states and agents acting on the states. However, progress in the system is modelled as a function mapping each state-agent pair to a set of possible resulting states. This consideration of sets of behaviours rather than simple traces allows for nondeterminism in the system. In contrast to the determinism approach, Bevier and Young give definitions in terms of a user's view of the system but leave the description

of such views as part of the development process for each individual system. Since there can be many different definitions for a user's view, the effectiveness of the approach will depend upon a suitable characterisation of the *View* function. This allows flexibility in defining the policies, but also introduces the possibility of insecurity if the views are not well-chosen. The approach provides a basic framework and any system description must be interpreted in terms of the model's components. The framework allows a nondeterministic system to be considered secure if the view of any low-level user could have arisen without high-level activity. However, it does not deal with information flow caused by observed refusals. The approach is also subject to the Refinement Paradox.

9.1.4 The deterministic security conditions

The background to the development of the deterministic security conditions was described in Chapters 1 and 2. The main attraction of these conditions is the persuasiveness of the theoretical link between nondeterminism and potential information flow as outlined by Roscoe [107]. Instead of trying to categorise and limit specific interactions as some approaches have done, the determinism conditions take a more abstract view. If the appearance of the system to a low-level user is completely deterministic then no actions of a high-level user can possibly have any effect. The way in which the appearance of the system to the low-level user is defined is crucial to the definition of security. The lazy and eager deterministic conditions are examples of this. Further discussion on ways of abstracting the high-level user's behaviour is given by Roscoe [109]Chapter 12 along with associated applications in the area of fault tolerance.

The deterministic conditions detect both existing interference from high to low and possible interference which might arise through injudicious resolution of under- specification. Hence these can be regarded as rejecting all specifications which are potentially insecure. Because of this, the Refinement Paradox does not arise for these conditions. A specification proved secure at the top level can be refined to an implementation with no further security

checks needed and with no security-related constraints on the refinement relation. Determinism is defined in a similar way in a variety of different notations and so the definitions are not limited to a single language or a specific approach. The conditions are stated in a concise and simple way and the use of FDR has shown that automatic verification is possible for CSP specifications.

Despite the theoretical advantages of the determinism properties it is still necessary to question their suitability and practicality for general use. As mentioned above, they have proved beneficial in the specific area of analysing security protocols. The case studies of Chapters 7 and 8 attempt to view the deterministic approach in the wider context of general systems development. When viewed in this way, some of the strengths of the approach also cause difficulties. The main problem is that the conditions are so strong that, in practice, no system could ever be expected to obey them. Even a limited application to a secure subsystem is unlikely to conform strictly to these definitions. This is a difficulty which must be faced by all the information flow properties, but is of additional significance for the even more limiting deterministic security conditions. It is certainly useful to have a theoretical basis for deciding what constitutes (within the bounds set) total absence of information flow. It has perhaps been thought in the past that total system security was achievable. However, it is now more generally acknowledged that no system which performs any useful work will be perfectly secure. The task of defining a security policy amounts to deciding what the unacceptable risks are. A useful security condition in this context is one which is flexible enough to capture a range of different requirements, many of which may fall short of the ideals of perfect security. It is possible for the deterministic security conditions to be modified to allow, for example, specific conditional policies, but they are still very rigid and incorporating a large number of exceptions may become unwieldy.

Another consequence of the strength of the deterministic security conditions is the difficulties encountered in applying them at an abstract level. The case study of Chapter 8 illustrates that it is very useful to allow nondetermin-

ism at the top level of specification. The deterministic security conditions may only be applied at the stage where all nondeterminism to the low-level user is removed. This conflicts with the desirability of proving security (and functional) properties of the system at as abstract a level as possible to make proofs more manageable. In some systems there may be genuine nondeterminism (such as the possible loss of messages between nodes of a network). Another example of this was seen in Chapter 8 where it was necessary at the top level to model queues of messages as bags. For a distributed refinement, it is not possible to give a specific ordering to the receipt of messages.

There may be many situations for which information flow analysis and hence the deterministic security properties are unsuitable. If covert channels are not an issue then simple access controls may be sufficient. There are also situations in which the complexity of the policy or the need for specialised requirements may weigh against the use of the deterministic security conditions. The use of action systems in no way limits us to a deterministic approach, and further applications are considered below in Section 9.3.1. As shown in the case study of Chapter 8, the combination of state and events in action systems allows a system to be viewed in a variety of ways. Also, different requirements pertaining to both aspects may be stated and proved.

9.2 Conclusions

This section lists the main points arising from the work of previous chapters, highlighting the contribution made by this research.

- The main contribution of the research is the definition of deterministic security properties for action systems. This is achieved by extending the existing theory of action systems with a definition of determinism which is shown to be equivalent to the CSP concept of determinism. Three basic security conditions are defined and these are again shown to be equivalent to their CSP counterparts. Using the results obtained here, it is possible to construct security policies and carry out the development of secure systems in a novel way using action systems. This work extends the results of Roscoe [107] and Butler [20].

- Using action systems has facilitated an approach which gives equal consideration to both state and the interaction of events within a system. Z has previously been commonly used in security specifications, but capturing security flow properties in a state-based language requires additional structure. The approach here accommodates the same detailed, structured specification of state, yet incorporates information about event interaction which can be used to define interference between users. This improves on the situation often found where some security aspects are ignored or, as with the Theta specification [54], where they are treated in an ad hoc fashion.

- The work here explores a pragmatic approach to the development of secure systems. The case studies show that action system specifications are suitable for use with reasonably large examples, allowing state to be structured in a number of ways (such as with Z schemas or specification statements) and combining descriptive clarity with the capacity for formal analysis of security properties. This again extends the work of Roscoe *et al.* [107, 110] where manipulation of complex state descriptions would be difficult.

- Action systems provide a formal marriage of state and events which, through simulation, allows both data and operation refinement to be carried out together. A state-based approach has often been preferred for general systems development, and this work allows the consideration of security to be brought into that process. CSP allows an algebraic approach, but this does not deal as directly with the state. Alternatively, Z is widely used for state-based specifications, but there are difficulties here in interpreting the actions as events. For example, interpretation of the preconditions will affect the security of the system. They could be viewed in a similar way as the guards of actions, but then the traditional weakening of preconditions through refinement would not preserve this correspondence. The approach here provides a simpler solution than, for example, the method of interpreting Z specifications

in terms of a secure system model [25]. Also, it does not rely on placing non-standard (and hard to verify) constraints on an existing language. Other key concepts such as the meaning of determinism for a Z specification also need to be made clear. These issues and the possible incorporation of Z operation schemas within action systems will be facilitated by the weakest precondition semantics for Z recently proposed by Cavalcanti and Woodcock [23].

- Proof of information flow properties can be difficult. The deterministic properties for action systems can be hard to verify directly. This thesis introduces some novel approaches to these proofs to make the task more manageable. The technique of defining a simple deterministic system of which the system under consideration is a refinement was used in Chapter 8. This looks promising since it reduces the information flow proofs to a proof of refinement. This is an area where previous work on action system refinement can be of assistance and for which tool support is likely to become available. The approach bears similarity to the FDR method of proving determinism by resolving nondeterministic choice and showing equivalence. It may be possible to exploit this technique to simplify the verification task for action systems.

- An approach such as this has clear advantages over a route which incorporates both state and events by translating between them, such as that demonstrated by Roscoe *et al.* [110]. A development method which requires translation steps is particularly prone to the introduction of errors. It introduces additional stages to the process and requires developers to be familiar with a number of different notations.

- Whilst confirming the benefits of the approach, the work here also identifies a number of difficulties. For the deterministic properties themselves, the greatest difficulty arises from the debarring of all nondeterminism from the specification. This limits the suitability of the definitions. For action systems, refinement conditions can be very difficult to prove. Further work is needed in this area.

- The combination of state and events allows for security policies to be formed from a variety of different components. For example, in Chap-

ter 8 a deterministic property for information flow is combined with restrictions on the state. This could be extended to incorporate other aspects of security, such as integrity and authentication. The need for this has been noted by a number of authors, for example Boswell [16].

- Use of the deterministic security properties avoids the problem of the Refinement Paradox which has continued to be a difficulty in the development of secure systems and is inherent in nearly all the definitions described in Chapter 2. If such a property can be proved at an abstract level there is no need for further validation after refinement. This thesis shows that such an approach is theoretically valid for state based systems also. The work here considers the practicalities of the approach as well as soundness of the theory. The case studies reveal that in practice it is often not possible to remove the nondeterminism from the top-level specification.

- The work here brings together results from a number of different areas, combining aspects of action systems, CSP, Z, specification statements and weakest precondition. The diversity of approach and notation in computer science can cause complication, as evidenced by the many different information flow properties reviewed in Chapter 2 and the difficulty of comparing them and applying them. Here, we bring notations together to combine some of the most useful aspects of each.

- The use of action systems allows parallel decomposition of a state-based specification, in which each component can be refined separately. The case studies have concentrated on various issues of networks. This area of application, where security considerations are currently of particular concern, is shown to be well- suited to an action system style of specification. The case studies suggest that action systems are a good choice for the state-based description of distributed systems.

- The case studies also show the importance of detailed representation of state. Chapter 7 considered the effect of encryption at various levels,

showing the consequences this has for the security of the system. For example, the layering of messages sent via a communications protocol can be expressed very clearly using the type system Z. The correct representation of state is thus vital to the security of the system. It would be possible to use a modified Z type-checker to provide a preliminary check on the specification.

- Defining the deterministic properties for action systems shows that the notation is well-suited to dealing with information flow properties. The notation does not restrict us to any one specific approach and other information flow properties can be defined as required. Indeed, since action systems deal directly with both state and events they provide an ideal notation in which to compare properties from a variety of different notations (see below).

9.3 Future work

The work of this thesis suggests a number of further directions for research, in both practical and theoretical aspects of security. This section identifies some of these.

9.3.1 Defining other security properties

The deterministic security conditions from CSP have been shown to be equally applicable to action systems. However, there are other information flow properties which we may wish to consider and these could be accommodated in action systems in a similar manner. For example, as described in Chapter 2, in [47] Graham-Cumming proposed a noninterference condition based on the user's inability to distinguish between different states of the system. The definition is in CSP, but to represent the user's view, the concept of global state is introduced based on equivalence classes formed by traces. The idea is common to other noninterference definitions too: after any trace, the state must appear the same to a low-level user as it would had

the trace contained no high-level actions.

In action systems, the explicit presence of the state allows a direct assault to be made on this sort of definition. However, there is no need to try to partition the state to achieve a view. In an action system, values of state variables are revealed by the enabling of actions. So for example, an action to output the value of a state component will be enabled only for the current value. This leads to the following definition of noninterference in which $tr\ |_X$ represents the trace tr with all actions from the set X removed.

Definition 39 *In an action system, the set of actions H is said to be noninterfering with the (disjoint) set L if for each trace tr, each $a_L \in L$ and each $a_H \in H$, $tr\ |_H$ is also a trace and:*

$$\overline{wp}(tr, gd\ a_L) = \overline{wp}(tr\ |_H, gd\ a_L)$$

That is, it is possible for a trace to enable a low-level action if and only if the same trace without H actions can also enable it. Although H and L must be disjoint they need not partition the actions of the system.

As in [47] it is possible to provide an unwinding theorem, giving sufficient conditions for Definition 39.

Theorem 14 *If for each trace tr, $a_H \in H$, $a_L \in L$ and $e \notin H$:*

$$\overline{wp}(a_H, gd\ a_L) = gd\ a_L$$

and

$$\overline{wp}(tr, gd\ a_L) = \overline{wp}(tr\ |_H, gd\ a_L)$$
$$\Rightarrow \overline{wp}(tr\hat{}\langle e\rangle, gd\ a_L) = \overline{wp}((tr\ |_H)\hat{}\langle e\rangle, gd\ a_L)$$

then H is noninterfering with L.

Proof by induction on length of traces.

This is a strictly weaker condition than the determinism one which entails that for a trace otr in the obscured system:

$$\overline{wp}(otr, gd\ a_L) = wp(otr, gd\ a_L)$$

Since high-level actions in the obscured system may skip at any point, their presence or absence can have no effect on enabling $gd\ a_L$. Hence, for the equivalent trace tr in the original system:

$$\begin{aligned}
&\overline{wp}(tr, gd\ a_L) \\
&= \overline{wp}(otr, gd\ a_L) \\
&= \overline{wp}(otr\ |_H, gd\ a_L) \\
&= \overline{wp}(tr\ |_L, gd\ a_L)
\end{aligned}$$

The property is not in general preserved by refinement.

There may well be uses for this and other information flow properties for which it would be useful to provide action system definitions. The large number of such properties makes it very difficult to see the similarities and differences between them and to compare their suitability for a particular purpose. A useful direction for future research would be to define the various properties in a common notation to allow this comparison and assessment to take place. As discussed above, some categorisation of results has already been attempted but this has been purely event-based and does not provide motivation for choice between the available properties. The use of action systems for this purpose would be an interesting area to explore and could provide a future doctoral research topic.

9.3.2 Security modelling

The work of Chapter 7 shows that system modelling for a secure system has moved away from the approach of a single policy (such as multi-level security) which can be applied throughout the whole system. Particularly in the area of network security, components are likely to be more diverse, with a move from system-wide security to a more individual view in which each user or local network is responsible for their own security. Users may decide to rely on a particular encryption algorithm or to receive assistance from other authorities on authentication, but the ultimate decisions are taken locally. This is an approach explored by Bull *et al.* [42]. With this view of the system, the way in which security is assessed will also be somewhat different. A clear

statement needs to be made about what constitutes confidentiality. In the same way that the development process for safety-critical systems includes a risk analysis, so the development of a secure system needs to include a phase of recognising possible illegal flows and deciding what (if anything) should be done. These issues are important ones for the development of secure systems and it would be of interest to develop them further. The use of action systems for this purpose is one possible approach, and it would be useful to investigate these aspects and their rôle in the overall development process.

9.3.3 Further examples

The case studies of Chapters 7 and 8 provide an introduction to the study of several aspects of network security. It would be useful to extend this work to consider other specific project developments in this area. Further work is needed to establish the practicality of action systems and the suitability of information flow properties. Practical applications are needed to test the theory and to show how it fits into the overall plan of system development. This type of work could form the basis for a larger-scale research project involving collaboration with industrial partners working on the development of secure systems.

9.3.4 Action systems and proof

One of the biggest problems with formal development techniques is the complexity of proofs. This is no exception for action systems, and the case studies of this thesis have shown that proof of determinism can be very difficult. The technique of reducing verification of determinism to that of a refinement helps the situation, but proof of refinement can itself be problematic. The difficulty is two-fold. Firstly, the refinement conditions in their general form are simple to state but inherently difficult to verify. Where simplified conditions apply, the situation is made much more tractable. Further work needs to be done in providing rules for action system refinement. The second problem

is that even relatively simple proofs are tedious by hand. Tool support is needed to make the development process feasible. There are several possible avenues to explore. Firstly, the B-tool has already been used by Walden and Sere [130] for refinement of the Back style of action systems. Butler has also explored the relationship of the CSP style of action systems with abstract machine notation [2]. So it is likely that use of the B-tool could be beneficial.

FDR has proved very successful with CSP verification, and it may be that ways can be found of representing state which make it applicable to general systems. However, because of the problems of state-space explosion it is not likely that such developments will occur in the near future.

Another option is the use of a refinement editor which could deal with actions written in the specification statement style. Such a tool is currently under development by a team based at Abo Akademi, Finland [21]. To be of most use, this needs to be able to deal with refinements of whole systems as well as of single actions. This, combined with a richer set of refinement rules, could support a refinement calculus approach in which specifications are developed through small, easily proved steps. Given the difficulty of proving large steps of refinement, this may well be a fruitful area to explore.

Appendix A

A CSP reference

Communicating sequential processes (CSP) were first described by Hoare [59]. Processes are defined in terms of the (atomic) events in which they participate, with a process definition outlining the possible sequences of events for that process. The set of events in which a process may engage is referred to as its alphabet. A trace of a process is a possible sequence of events for that process. Processes communicate by synchronising on common events in their alphabets. If an infinite sequence of internal events can occur the process is said to diverge.

The following CSP notation is used in this thesis. For any processes P and Q, and set of events A:

$\alpha(P)$	the alphabet of P, that is, the set of events in which P may participate
A^*	the set of all finite sequences of elements of A
A^ω	the set of all infinite sequences of elements of A
$traces(P)$	the set of finite traces of P
$refusals(P)$	the set of events of $\alpha(P)$ in which P may decline to engage initially
$fails(P)$	the set of pairs (tr, R) where tr is a trace after which all events in R may be refused
$divs(P)$	the set of traces of P after which P may diverge
$P =_T Q$	trace equivalence of processes P and Q
$P =_{FD} Q$	failures-divergence equivalence of processes P and Q

$a \rightarrow P$	event a, then P
$a : A \rightarrow P$	the (external) choice of event a from A, then P
P *deterministic*	P is non-divergent and cannot refuse any events in which it might engage, that is, for each trace tr and each $x \in \alpha(P)$: $\mathit{divs}(P) = \emptyset \wedge$ $tr^\frown\langle x\rangle \in \mathit{traces}(P) \Rightarrow (tr, \{x\}) \notin \mathit{fails}(P)$
$P \mid\mid\mid Q$	the interleaving operator which joins processes to operate concurrently without direct synchronisation. The traces of $P \mid\mid\mid Q$ consist of all arbitrary interleavings of P and Q
$P \setminus A$	the hiding operator which conceals all occurrences of events in A. Traces of $P \setminus A$ are traces of P with A events removed. Hidden events are thought of as taking place internally whenever the possibility arises
RUN_A	the process always ready to participate in any event from H: $RUN_A = a : A \rightarrow RUN_A$
P/t	the process which behaves as P after engaging in trace t
$P \parallel Q$	the parallel combination of processes P and Q synchronising on events from $\alpha(P) \cap \alpha(Q)$
$P \underset{A}{\parallel} Q$	the parallel combination of processes P and Q synchronising on events from A.
$P \Box Q$	external choice between P and Q
$P \sqcap Q$	nondeterministic choice between P and Q
$STOP_A$	the process with alphabet A which refuses to participate in any event.
$HAVOC_A$	the most nondeterministic divergence-free process: $HAVOC_H = STOP \sqcap (x : H \rightarrow HAVOC_H)$
$FINITE_A$	similar to $HAVOC_A$ but with no infinite traces. It may be defined: $FINITE_A = \sqcap\{Q_n \mid n \in \mathbb{N}\}$ where $Q_0 = STOP$ and $Q_{n+1} = a : A \rightarrow Q_n$
$t \upharpoonright A$	trace t with all the events of set A removed

P^0	the set of events in which P may engage initially

Appendix B

Weakest precondition and basic notation

Weakest precondition and the guarded command language

The *weakest precondition* for statement S to establish the condition *post* is written:

$$wp(S, post)$$

It is the predicate describing the set of states in which execution of S is guaranteed to terminate in a state satisfying post.

A *predicate* may be viewed as a set of states. In the context of some non-empty set of states Σ, predicate α is a subset of Σ. For two predicates α and β to be equivalent, written $\alpha \equiv \beta$ the two sets of states must be equal. Predicate α entails predicate β, written $\alpha \Rrightarrow \beta$, if the states of α form a subset of the states of β.

A function from predicates to predicates is referred to as a *predicate transformer*. Predicate transformer f is *monotonic* if whenever $\alpha \Rrightarrow \beta$ then it follows that $f(\alpha) \Rrightarrow f(\beta)$.

If f is a predicate transformer then predicate X is a *fixed point* of f if:

$$f(X) \equiv X$$

The *least fixed point* of f is denoted:

$$\mu X \bullet f(X)$$

That the least fixed point exists for monotonic f for the set of all predicates with entailment ordering $\Rrightarrow$ is a consequence of the Knaster-Tarski Theorem [129].

For predicate α the expression $\alpha[e/x]$ denotes substitution of e for each free occurrence of x in α.

Dijkstra's guarded command language [31] defines a programming language in terms of the weakest precondition of each construct. The basic commands used here are defined as follows.

$$wp(skip, \alpha) = \alpha$$

$$wp(abort, \alpha) = false$$

$$wp(s1;\ s2, \alpha) = wp(s1, wp(s2, \alpha))$$

$$wp(x := e, \alpha) = \alpha[e/x] \qquad \text{for well-defined } e$$

The alternative command is written:

if $g_1 \rightarrow c_1$
$[]$ $g_2 \rightarrow c_2$
$\vdots$
$[]$ $g_n \rightarrow c_n$
fi

If this is denoted by IF then it can be defined for well-defined g_i:

$$wp(IF, \alpha) = (\exists\, i : 1\,..\,n \bullet g_i) \wedge (\forall\, i : 1\,..\,n \bullet g_i \Rightarrow wp(c_i, \alpha))$$

In addition to the basic assignment statement, nondeterministic assignment is also used. If S is a set then $x :\in S$ assigns x some value of S. Its weakest precondition is:

$$wp(x :\in S, \alpha) = s \neq \emptyset \wedge (\forall\, s : S \bullet \alpha[s/x])$$

Similarly,

$$wp(x :\subseteq S, \alpha) = (\forall\, t : \mathbf{P}\ S \bullet \alpha[t/x])$$

Specification statements

A specification statement consists of a precondition *pre*, a postcondition *post* and a frame x of variables. It is written:

$$x : [pre, post]$$

From a state satisfying *pre* the statement assigns values to x to establish *post*. Occurences of x_0 in *post* refer to the original values of x. The weakest precondition of the specification statement is:

$$wp(x : [pre, post], \alpha) \equiv pre \wedge (\forall x \bullet post \Rightarrow \alpha)[x/x_0])$$

So, for example, the guard of action:

$$x : [s \neq \emptyset, x \in s] \qquad (1)$$

is *true*. Note that while weakening the precondition represents a valid refinement for a specification statement it is not sufficient for action system refinement. This is because action systems are also concerned with possible sequences of actions. So, for example, since the guard of:

$$x : [true, x \in s] \qquad (2)$$

is $s \neq \emptyset$, action (2) could not be used to refine action (1) because the condition on guards would not be met.

A useful variation is the specification with precondition omitted:

$$x : [post]$$

This is used by both Morgan [94] and Butler [20] but their definitions differ. We adopt the approach of Butler [20] where :

$$x : [post] \mathrel{\widehat{=}} x : [true, post]$$

since this is usually the most convenient interpretation for use with action system specifications. It has the useful property that:

$$wp(x : [post], \alpha) \equiv (\forall x \bullet post \Rightarrow \alpha)[x/x_0]$$

and hence:

$$gd(x : [post]) \equiv (\exists\, x \bullet post)[x/x_0]$$

For a system with invariant INV we follow the convention of allowing input action:

action a **in** $i?$: T :–
$x : [post]$

to represent:

$$x : [x \in T \wedge INV_0 \wedge INV \wedge post]$$

and allowing output action:

action b **out** $o!$: T :–
$o!, x : [post]$

to represent:

$$o!, x : [o! \in T \wedge INV_0 \wedge INV \wedge post]$$

where:

$INV_0 \equiv INV[v_0/v]$ for state variable v

Appendix C

A Z reference

Z [125] is a specification notation which describes the state of a system explicitly. Operations are defined by describing their effect on the system state. Z encompasses a range of (mainly standard) mathematical and logical notation for expressing the typed set theory used in its definitions. It also makes use of a structuring mechanism, known as the schema calculus, which allows the description of both state and operations to be built up in a modular fashion. Z and its use are described in a number of text books (see [104, 125, 133] for example). The list given below summarises the notation used in this thesis.

Sets and relations

$[T]$	introduces T as a basic type of the specification
$\mathbb{P}\ X$	the power set of the set X, that is, the set of all subsets of X
$\emptyset$	emptyset
$x \in X$	x is an element of set X
$X \subseteq Y$	set X is a subset of set Y
$X \subset Y$	set X is a proper subset of set Y
$X - Y$	set X minus set Y
$X \cup Y$	the union of sets X and Y
$X \cap Y$	the intersection of sets X and Y
$\bigcup X$	the generalised union of all sets in X
$\bigcap X$	the generalised intersection of all sets in X

$X \times Y$	the Cartesian product of sets X and Y, that is the set of all pairs, (x, y), with $x \in X$ and $y \in Y$
$\mathbb{N}$	the set of all natural numbers, $\{0, 1, 2, \ldots\}$
$\mathbb{N}_1$	the set of all positive natural numbers, $\{1, 2, \ldots\}$
$(1 \,..\, n)$	the set of all numbers from 1 to n inclusive
$x?$	a variable name ending in ? represents an input
$y!$	a variable name ending in ! represents an output
x'	a variable name ending with a dash represents the state of the variable after an operation has been performed on it
$X \leftrightarrow Y$	the set of all relations between X and Y, where each relation is a set of ordered pairs: $X \leftrightarrow Y == \mathbb{P}(X \times Y)$
$x \mapsto y$	x maps to y in some relation; alternative notation for (x, y)
$\mathrm{dom}\, R$	the domain of the relation R defined for $R : X \leftrightarrow Y$ as: $\{x : X \mid (\exists y : Y \bullet x \mapsto y \in R)\}$
$\mathrm{ran}\, R$	the range of the relation R defined for $R : X \leftrightarrow Y$ as: $\{y : Y \mid (\exists x : X \bullet x \mapsto y \in R)\}$
R^*	the reflexive transitive closure of the relation R: if R^n represents the iteration of R n times, then: $R^* = \bigcup \{n : \mathbb{N} \bullet R^n\}$
$X \nrightarrow Y$	the set of all partial functions from X to Y defined: $\{R : X \leftrightarrow Y \mid (\forall x : X;\ y, z : Y \bullet x \mapsto y \in R \land x \mapsto z \in R \Rightarrow y = z)\}$
$X \rightarrow Y$	the set of all total functions from X to Y defined: $\{f : X \nrightarrow Y \mid \mathrm{dom} f = X\}$

Logical expressions

$\neg P$	negation of P
$P \land Q$	conjunction of P and Q
$P \lor Q$	disjunction of P and Q
$P \Rightarrow Q$	P implies Q
$\forall x : T \mid P \bullet Q$	universal quantification - equivalent to:

$$\forall x : T \mid P \Rightarrow Q$$

$\exists x : T \mid P \bullet Q$ existential quantification - equivalent to:

$$\exists x : T \mid P \wedge Q$$

Sequences and bags

$\mathrm{seq}\, X$	the set of all finite sequences of elements of X:
$\mathrm{iseq}\, X$	the set of all injective sequences of elements of X: that is, no element occurs more than once
$\langle x_1 \ldots x_n \rangle$	the sequence with elements x_1 to x_n
$s1 \frown s2$	concatenation of sequences $s1$ and $s2$
last s	the last element of the non-empty sequence s
front s	all but the last element of the non-empty sequence s
$\mathrm{bag}\, X$	the set of all bags of elements of X: each element in the bag may occur a finite number of times, thus: $\mathrm{bag}\, X == X \pfun \mathbb{N}_1$
$\lbag x_1 \ldots x_n \rbag$	the bag with elements x_1 to x_n
x *in* b	x is an element of bag b
$b1 + b2$	addition of bags $b1$ and $b2$
$b1 - b2$	subtraction of bag $b2$ from bag $b1$

Schemas

A Z schema is a named unit bringing together a collection of variable declarations and a predicate relating the variables. For example, a schema $S1$ may be defined:

$S1$

$x : \mathbb{N}$

$y : \mathbb{P}\, \mathbb{N}$

$x \in y$

The name of a previously-defined schema may be included in a schema definition, in which case the declarations and predicate will be incorporated. For instance:

$S2$

$S1$

$z : \mathbb{N}$

$z \notin y$

is equivalent to:

$S2$

$x : \mathbb{N}$

$y : \mathbb{P}\ \mathbb{N}$

$z : \mathbb{N}$

$(x \in y) \land (z \notin y)$

Free type definitions

A free type definition defines a new named type from the existing types of the specification. For example, if *COLOUR* and *SHAPE* are existing types of a specification then:

$$PROPERTY ::= col\langle\langle COLOUR\rangle\rangle \mid sha\langle\langle SHAPE\rangle\rangle$$

describes an overall type, *PROPERTY*, constructed from the two existing ones.

Orderings

A partial order is a relation $R : X \leftrightarrow X$ which is:

reflexive	$\forall x : X \bullet x\ R\ x$
antisymmetric	$\forall x, y : X \bullet x;\ R\ y \land y\ R\ x \Rightarrow x = y$
transitive	$\forall x, y, z : X \bullet x\ R\ y \land y\ R\ z \Rightarrow x\ R\ z$

A linear order is a partial order such that:

$$\forall x, y : X \bullet x \; R \; y \vee y \; R \; x$$

Appendix D

Proofs

The One Point Rules

The One Point Rules are logical laws which are useful in the type of proofs carried out here. The first of these is for existentially quantified formulae. If part of the quantified statement gives an exact value for the quantified variable, then the quantification can be removed, replacing the variable wherever it appears.

The $\exists$ One Point Rule

$$(\exists x : S \bullet p \wedge x = t) \equiv t \in S \wedge p[t/x] \qquad [x \text{ n.f.i. } t]$$

The side condition is the requirement that x is not free in t. A similar rule holds for universally quantified expressions.

The $\forall$ One Point Rule

$$(\forall x : S \bullet x = t \wedge p) \equiv t \in S \wedge p[t/x] \qquad [x \text{ n.f.i. } t]$$

Again, the side condition that x is not free in t is required. A variant on this is the equivalence:

$$(\forall x : S \bullet x = t \Rightarrow p) \equiv t \notin S \vee p[t/x] \qquad [x \text{ n.f.i. } t]$$

This follows from the $\exists$ One Point Rule.

Proofs for Chapter 6

Proof of refinement for Example 47

Each condition of Definition 34 is checked in turn.

1. For the left hand side (LHS):

$$wp(s := \emptyset;\ n :\in \mathbf{N}, \alpha) \equiv (\forall n : \mathbf{N} \bullet \alpha[\emptyset/s])$$

For the right hand side (RHS):

$$wp(m := 0, (\exists s : \mathbf{P}\ \mathbf{N};\ n : \mathbf{N} \bullet s = (1 \mathinner{..} m) \wedge n = m \wedge \alpha))$$
$$\equiv wp(m := 0, \alpha[(1 \mathinner{..} m), m/s, n]) \qquad \text{[One point rule]}$$
$$\equiv \alpha[\emptyset, 0/s, n] \qquad \text{[}m \text{ not free in } \alpha\text{]}$$

$LHS \equiv\!\!\!\Rightarrow RHS$, so condition 1 holds.

2. Considering action $a1$:

$$LHS \equiv (\exists s : \mathbf{P}\ \mathbf{N};\ n : \mathbf{N} \bullet s = (1 \mathinner{..} m) \wedge n = m \wedge wp(\mathcal{A}17_{a1}, \alpha))$$
$$\equiv (\exists s : \mathbf{P}\ \mathbf{N};\ n : \mathbf{N} \bullet s = (1 \mathinner{..} m) \wedge n = m \wedge s \neq \mathbf{N} \wedge$$
$$(\forall n : (\mathbf{N} - s) \bullet \alpha[s \cup \{n\}/s])) \qquad \text{[Defn. of } wp\text{]}$$
$$\equiv (\forall n : (\mathbf{N} - (1 \mathinner{..} m)) \bullet \alpha[(1 \mathinner{..} m) \cup \{n\}/s]) \qquad \text{[One point rule]}$$

$$RHS \equiv wp(\mathcal{B}2_{a1}, (\exists s : \mathbf{P}\ \mathbf{N};\ n : \mathbf{N} \bullet s = (1 \mathinner{..} m) \wedge n = m \wedge \alpha))$$
$$\equiv wp(m := m + 1, \alpha[(1 \mathinner{..} m), m/s, n]) \qquad \text{[One point rule]}$$
$$\equiv \alpha[(1 \mathinner{..} m + 1), m + 1/s, n] \qquad \text{[}m \text{ not free in } \alpha\text{]}$$

$LHS \equiv\!\!\!\Rightarrow RHS$ since $m + 1 \in (\mathbf{N} - (\mathrm{ran}\ t))$, so condition 2 holds for $a1$.
Similarly for $a2$:

$$LHS \equiv (\exists s : \mathbf{P}\ \mathbf{N};\ n : \mathbf{N} \bullet s = (1 \mathinner{..} m) \wedge n = m \wedge wp(\mathcal{A}17_{a2}, \alpha))$$
$$= (\exists s : \mathbf{P}\ \mathbf{N};\ n : \mathbf{N} \bullet s = (1 \mathinner{..} m) \wedge n = m \wedge$$
$$(s \neq \emptyset \Rightarrow (\forall n : s \bullet \alpha[s - \{n\}/s]))) \qquad \text{[Defn. of } wp\text{]}$$
$$\equiv (1 \mathinner{..} m) \neq \emptyset \Rightarrow (\forall n : (1 \mathinner{..} m) \bullet \alpha[(1 \mathinner{..} m) - \{n\}/s]) \text{[One point rule]}$$

$$RHS \equiv wp(\mathcal{B}2_{a2}, (\exists\, s : \mathbf{P}\ \mathbf{N};\ n : \mathbf{N} \bullet s = (1\,..\,m) \wedge n = m \wedge \alpha))$$
$$\equiv m \neq 0 \Rightarrow wp(m := m - 1, \alpha[(1\,..\,m), m/s, n]) \qquad \text{[One point rule]}$$
$$\equiv m \neq 0 \Rightarrow \alpha[(1\,..\,m-1), m-1/s, n] \qquad \text{[}m \text{ not free in } \alpha\text{]}$$

$LHS \not\equiv RHS$ since $m - 1 \in (1\,..\,m)$. So Condition 2 holds for $a2$.

3. Finally we consider the guards. For $a1$:

$$(\exists\, s : \mathbf{P}\ \mathbf{N};\ n : \mathbf{N} \bullet s = (1\,..\,m) \wedge n = m \wedge true)$$
$$\equiv true$$

For $a2$:

$$(\exists\, s : \mathbf{P}\ \mathbf{N};\ n : \mathbf{N} \bullet s = (1\,..\,m) \wedge n = m \wedge s \neq \emptyset)$$
$$\equiv (1\,..\,m) \neq \emptyset$$
$$\equiv m \neq 0$$

Proof of refinement for Example 48

The conditions of Definition 34 are checked.

1. This is the initialisation condition.

$$LHS \equiv wp(s1 := no;\ k1 :\in KEY, \alpha)$$
$$\equiv (\forall\, k1 : KEY \bullet \alpha[no/s1])$$

$$RHS \equiv wp(s2 := no;\ k2 :\in KEY, (\exists\, k1 : KEY;\ s1 : STATUS \bullet R \wedge \alpha))$$
$$\equiv (\exists\, k1 : KEY;\ s1 : STATUS \bullet s1 = no \wedge \alpha) \qquad \text{[}s2, k2 \text{ n.f.i. } \alpha\text{]}$$
$$\equiv (\exists\, k1 : KEY \bullet \alpha[no/s1]) \qquad \text{[One pt. rule]}$$

Therefore, since KEY is assumed nonempty: $LHS \not\equiv RHS$.

2. For *startsession*:

$$LHS \equiv (\exists\, k1 : KEY;\ s1 : STATUS \bullet R \wedge$$

$$wp(s1 = no \rightarrow s1 := yes, \alpha))$$

$$\equiv (\exists k1 : KEY;\ s1 : STATUS \bullet (s1 = s2) \wedge$$
$$(s1 = s2 = send \Rightarrow k1 = k2) \wedge (s1 = no \Rightarrow \alpha[yes/s1]))$$

$$\equiv (\exists k1 : KEY \bullet (s2 = send \Rightarrow k1 = k2) \wedge$$
$$(s2 = no \Rightarrow \alpha[yes/s1]))$$

$$RHS \equiv wp(s2 = no \rightarrow s2 := yes;\ k2 :\in KEY,$$
$$(\exists k1 : KEY;\ s1 : STATUS \bullet R \wedge \alpha))$$

$$\equiv s2 = no \Rightarrow (\exists k1 : KEY;\ s1 : STATUS \bullet s1 = yes \wedge \alpha)$$

$$\equiv s2 = no \Rightarrow (\exists k1 : KEY \bullet \alpha[yes/s1])$$

Therefore $LHS \equiv\!\!\!\Rightarrow RHS$.

For *sendkey*:

$$LHS \equiv (\exists k1 : KEY;\ s1 : STATUS \bullet R \wedge$$
$$wp(s1 = yes \rightarrow k1 :\in KEY;\ s1 := send, \alpha))$$

$$\equiv (\exists k1 : KEY;\ s1 : STATUS \bullet R \wedge$$
$$(s1 = yes \Rightarrow (\forall k1 : KEY \bullet \alpha[send/s1])))$$

$$\equiv (\exists k1 : KEY \bullet (s2 = send \Rightarrow k1 = k2) \wedge$$
$$(s2 = yes \Rightarrow (\forall k1 : KEY \bullet \alpha[send/s1])))$$

$$RHS \equiv wp(s2 = yes \rightarrow s2 := send,$$
$$(\exists k1 : KEY;\ s1 : STATUS \bullet R \wedge \alpha))$$

$$\equiv s2 = yes \rightarrow$$
$$(\exists k1 : KEY;\ s1 : STATUS \bullet s1 = send \wedge k1 = k2 \wedge \alpha)$$

$$\equiv s2 = yes \Rightarrow \alpha[send, k2/s1, k1])$$

Again, since KEY is nonempty: $LHS \equiv\!\!\!\Rightarrow RHS$.

3. Finally the condition on guards must be checked for each action.

For *startsession*:

$$LHS \equiv (\exists k1 : KEY;\ s1 : STATUS \bullet R \wedge s1 = no)$$
$$\equiv (\exists k1 : KEY \bullet s2 = no)$$
$$\equiv s2 = no$$
$$\equiv RHS$$

and similarly for *sendkey*.

All the refinement conditions are satisfied and hence:

$$KeyServer1 \sqsubseteq_R KeyServer2$$

Proof of refinement for Example 49

The conditions for Definition 35 are checked.
1. Initialisation:

$$LHS \equiv wp(s2 := no;\ k2 :\in KEY, \alpha)$$
$$\equiv (\forall k2 : KEY \bullet \alpha[no/s2])$$

$$RHS$$
$$\equiv wp(s1 := no;\ k1 :\in KEY, (\forall k2 : KEY;\ s2 : STATUS \bullet R \Rightarrow \alpha))$$
$$\equiv (\forall k1 : KEY \bullet (\forall k2 : KEY;\ s2 : STATUS \bullet s2 = no \Rightarrow \alpha))$$
$$\equiv (\forall k2 : KEY \bullet \alpha[no/s2]) \qquad [\forall \text{ One pt. rule, } k1 \text{ n.f.i. } \alpha]$$

Therefore $LHS \equiv RHS$.

2. For *startsession*:

$$LHS \equiv (\forall k2 : KEY;\ s2 : STATUS \bullet R \Rightarrow$$
$$wp(s2 = no \rightarrow s2 := yes;\ k2 :\in KEY, \alpha))$$
$$\equiv (\forall k2 : KEY;\ s2 : STATUS \bullet R \Rightarrow$$
$$(s2 = no \Rightarrow (\forall k2 : KEY \bullet \alpha[yes/s2])))$$

$$\equiv (\forall k2 : KEY;\ s2 : STATUS \bullet$$
$$(s1 = no \wedge s2 = no) \Rightarrow (\forall k2 : KEY \bullet \alpha[yes/s2]))$$
$$\equiv (\forall k2 : KEY;\ s2 : STATUS \bullet$$
$$s2 = no \Rightarrow (s1 = no \Rightarrow (\forall k2 : KEY \bullet \alpha[yes/s2])))$$
$$\equiv s1 = no \Rightarrow (\forall k2 : KEY \bullet \alpha[yes/s2])$$

$$RHS \equiv wp(s1 = no \rightarrow s1 := yes,$$
$$(\forall k2 : KEY;\ s2 : STATUS \bullet R \Rightarrow \alpha))$$
$$\equiv s1 = no \Rightarrow (\forall k2 : KEY;\ s2 : STATUS \bullet s2 = yes \Rightarrow \alpha))$$
$$\equiv s1 = no \Rightarrow (\forall k2 : KEY \bullet \alpha[yes/s2])$$

Therefore $LHS \equiv RHS$.

For *sendkey*:

$$LHS \equiv (\forall k2 : KEY;\ s2 : STATUS \bullet R \Rightarrow$$
$$wp(s2 = yes \rightarrow s2 := send, \alpha))$$
$$\equiv (\forall k2 : KEY;\ s2 : STATUS \bullet ((s1 = s2) \wedge$$
$$(s1 = s2 = send \Rightarrow k1 = k2)) \Rightarrow$$
$$(s2 = yes \Rightarrow \alpha[send/s2]))$$
$$\equiv (\forall k2 : KEY \bullet (s1 = send \Rightarrow k1 = k2) \wedge (s1 = yes \Rightarrow \alpha[send/s2]))$$
$$\equiv (\forall k2 : KEY \bullet s1 = send \Rightarrow k1 = k2) \wedge$$
$$(\forall k2 : KEY \bullet s1 = yes \Rightarrow \alpha[send/s2])$$
$$RHS \equiv wp(s1 = yes \rightarrow k1 :\in KEY;\ s1 := send,$$
$$(\forall k2 : KEY;\ s2 : STATUS \bullet R \Rightarrow \alpha))$$
$$\equiv s1 = yes \Rightarrow wp(k1 :\in KEY, (\forall k2 : KEY;\ s2 : STATUS \bullet$$
$$(s2 = send \wedge k1 = k2) \Rightarrow \alpha))$$
$$\equiv s1 = yes \Rightarrow (\forall k1 : KEY \bullet (\forall k2 : KEY;\ s2 : STATUS \bullet$$
$$(s2 = send \wedge k1 = k2) \Rightarrow \alpha))$$
$$\equiv s1 = yes \Rightarrow (\forall k1 : KEY \bullet \alpha[send, k1/s2, k2])$$

$$\equiv s1 = yes \Rightarrow (\forall k2 : KEY \bullet \alpha[send/s2])$$

Therefore $LHS \not\equiv RHS$.

3.For backwards simulation the condition on guards requires each subset of guards to be checked. Here there are 4 possible sets:

$$\{\}, \{startsession\}, \{sendkey\}, \{startsession, sendkey\}$$

For $\{\}$:

$$LHS \equiv (\forall k2 : KEY;\ s2 : STATUS \bullet R \Rightarrow gd(KeyServer2_{\{\}}))$$
$$\equiv (\forall k2 : KEY;\ s2 : STATUS \bullet R \Rightarrow false)$$
$$\equiv (\forall k2 : KEY;\ s2 : STATUS \bullet \neg R)$$
$$\equiv false$$
$$\equiv gd(KeyServer1_{\{\}})$$
$$\equiv RHS$$

Therefore $LHS \not\equiv RHS$.

For $\{startsession\}$:

$$LHS$$
$$\equiv (\forall k2 : KEY;\ s2 : STATUS \bullet R \Rightarrow gd(KeyServer2_{\{startsession\}}))$$
$$\equiv (\forall k2 : KEY;\ s2 : STATUS \bullet$$
$$((s1 = s2) \wedge (s1 = s2 = send \Rightarrow k1 = k2)) \Rightarrow s2 = no)$$
$$\equiv (\forall k2 : KEY \bullet (s1 = send \Rightarrow k1 = k2) \Rightarrow s1 = no)$$
$$\equiv (\exists k2 : KEY \bullet s1 = send \Rightarrow k1 = k2) \Rightarrow s1 = no$$
$$\equiv (s1 = send \Rightarrow \exists k2 : KEY \bullet k1 = k2) \Rightarrow s1 = no$$
$$\equiv (s1 = send \Rightarrow true) \Rightarrow s1 = no$$
$$\equiv s1 = no$$
$$\equiv gd(KeyServer1_{\{startsession\}})$$
$$\equiv RHS$$

A similar argument holds for the remaining two sets.

So with the rules for backwards simulation:

$$KeyServer2 \sqsubseteq KeyServer1$$

Proof of refinement for Example 50

Checking the conditions of Definition 36: // 1. Initialisation:

$$LHS \equiv wp(AbsCount_i, \alpha) \equiv wp(skip, \alpha) \equiv \alpha$$

$$RHS \equiv wp(ConcCount_i \ * \ \{reset\}, \alpha)$$
$$\equiv wp(ConcCount_i;\ IT_{reset}, \alpha)$$
$$\equiv wp(ConcCount_i;\ (skip \parallel reset), \alpha)$$
$$\equiv wp(ConcCount_i;\ skip, \alpha) \qquad \text{[Removing miraculous branches]}$$
$$\equiv wp(daysleft := 100, \alpha)$$
$$\equiv \alpha[100/daysleft] \equiv \alpha \qquad [daysleft \text{ n.f.i. } \alpha]$$

Therefore $LHS \equiv\!\!\!\Rightarrow RHS$.

2. Correctness:

$$LHS \equiv wp(true \rightarrow skip, \alpha) \equiv \alpha$$

$$RHS \equiv wp(daysleft > 0 \rightarrow daysleft := daysleft - 1 \ * \ \{reset\}, \alpha)$$
$$\equiv wp((daysleft > 0 \rightarrow daysleft := daysleft - 1);$$
$$(skip \parallel daysleft = 0 \rightarrow daysleft := 100), \alpha)$$
$$\equiv (daysleft > 0 \Rightarrow \alpha[daysleft - 1/daysleft]) \wedge$$
$$(daysleft = 1 \Rightarrow \alpha[100/daysleft])$$
$$\equiv daysleft > 0 \Rightarrow \alpha \qquad [daysleft \text{ n.f.i. } \alpha]$$

Therefore $LHS \equiv\!\!\!\Rightarrow RHS$.

3. Applicability:

$$LHS \equiv true$$

$$RHS \equiv gd_{\{reset\}}(daysleft > 0 \rightarrow daysleft := daysleft - 1)$$
$$\equiv \overline{wp}(IT_{\{reset\}};\ daysleft > 0 \rightarrow daysleft := daysleft - 1, true)$$
$$\equiv \overline{wp}(skip \parallel reset, daysleft > 0)$$
$$\equiv daysleft > 0 \vee daysleft = 0$$
$$\equiv true$$

So:

$$AbsCount \sqsubseteq_R ConcCount$$

Proof of refinement for Example 51

1. For the initialisation:

$$LHS \equiv wp(MultiStack_i, \alpha)$$
$$\equiv (\forall S \bullet \text{ran } stack = \{\langle\rangle\} \Rightarrow \alpha)$$

$$RHS \equiv wp(ConcStack_i, (\exists S \bullet R \wedge \alpha))$$
$$\equiv (\forall CS \bullet \text{ran } cclass = CLASS \wedge \text{ran } cstack = \{\langle\rangle\} \Rightarrow$$
$$(\exists S \bullet (\forall cl : CLASS \bullet stack\ cl = cstack\ (cclass^{-1}\ cl)) \wedge \alpha))$$
$$\equiv (\exists S \bullet (\forall cl : CLASS \bullet stack\ cl = \langle\rangle \wedge \alpha))$$
$$\equiv (\exists S \bullet \text{ran } stack = \{\langle\rangle\} \wedge \alpha)$$

If *empstack* denotes the abstract stack system with the stack for each classification empty, then, by the One Point Rules, both branches above are equivalent to $\alpha[empstack/stack]$ and so $LHS \equiv RHS$.

2. Correctness for *push*: for inputs $iu : USER$; $ic : CLASS$; $id : DATA$ the abstract *push* action is:

$$(clear\ iu) \leq ic \rightarrow (stack\ ic) := (stack\ ic)^\frown\langle id\rangle$$

output	action
error	$(clear\ iu) < ic \rightarrow skip$
empty	$((clear\ iu) < ic) \wedge ((stack\ ic) = \langle\rangle) \rightarrow skip$
$Rep(d)$	$((clear\ iu) < ic) \wedge ((stack\ ic) \neq \langle\rangle) \wedge (last(stack\ ic)) = d) \rightarrow skip$

Figure D.1 Abstract *top* actions for specific outputs

and the concrete version is:

$$(clear\ iu) \leq ic \rightarrow (cstack(cclass^{-1}\ ic)) := (cstack(cclass^{-1}\ ic))^\frown\langle id\rangle$$

Using these:

$$LHS \equiv (clear\ iu) \leq ic \Rightarrow \alpha[(cstack(cclass^{-1}\ ic))/(stack\ ic)]$$
$$\equiv RHS$$

For *pop* the reasoning is similar with both *LHS* and *RHS* equivalent to:

$$(clear\ iu) \leq ic \Rightarrow$$
$$(((cstack(cclass^{-1}\ ic)) = \langle\rangle \Rightarrow \alpha) \wedge$$
$$((cstack(cclass^{-1}\ ic)) \neq \langle\rangle \Rightarrow$$
$$\alpha[front(cstack(cclass^{-1}\ ic))/(cstack(cclass^{-1}\ ic))]))$$

Finally, for the bi-directional channel *top* with inputs $iu : USER$, $ic : CLASS$ the individual actions depend on the outputs in each case. Figure D.1 shows this for the abstract specification. The concrete actions follow a similar pattern and each is equivalent via the retrieve relation to its abstract counterpart. Therefore $LHS \equiv\!\!\rangle RHS$.

3. For each action the *LHS* of the guard condition is equivalent to the *RHS*. This shows that:

$$MultiStack \sqsubseteq ConcStack$$

Proof of refinement for Example 52

There are no internal actions and so the basic refinement rule of Definition 34 can be used.

1. Initialisation.

$$LHS \equiv wp(sess := null, \alpha) \equiv \alpha[null/sess]$$

$$\begin{aligned}
&RHS \\
&\equiv wp(psess, Asess := null, null, (\exists\, sess : STATUS;\ k : KEY \bullet R \wedge \alpha)) \\
&\equiv (\exists\, sess : STATUS;\ k : KEY \bullet k = pk \wedge sess = null \wedge \alpha) \\
&\qquad\qquad [psess, Asess \text{ n.f.i. } \alpha] \\
&\equiv \alpha[null, pk/sess, k]
\end{aligned}$$

Therefore: $LHS \equiv\!\!\Rightarrow RHS$

2. For *startsess*:

$$\begin{aligned}
LHS &\equiv (\exists\, sess : STATUS;\ k : KEY \bullet \\
&\qquad R \wedge wp(sess = null \rightarrow sess := ready, \alpha)) \\
&\equiv (\exists\, sess : STATUS;\ k : KEY \bullet R \wedge (sess = null \Rightarrow \alpha[ready/sess])) \\
&\equiv (\exists\, sess : STATUS;\ k : KEY \bullet (R \wedge sess = null \wedge \alpha[ready/sess]) \vee \\
&\qquad\qquad (R \wedge sess \neq null)) \\
&\equiv (\exists\, sess : STATUS;\ k : KEY \bullet R \wedge sess = null \wedge \alpha[ready/sess]) \vee \\
&\quad (\exists\, sess : STATUS;\ k : KEY \bullet R \wedge sess \neq null) \\
&\equiv (\exists\, sess : STATUS;\ k : KEY \bullet k = pk \wedge sess = null \wedge psess = null \\
&\qquad\qquad \wedge Asess = null \wedge \alpha[ready/sess]) \vee \\
&\quad (\exists\, sess : STATUS;\ k : KEY \bullet R \wedge sess \neq null) \\
&\equiv (psess = null \wedge Asess = null \wedge \alpha[ready, pk/sess, k]) \vee \\
&\quad (\exists\, sess : STATUS;\ k : KEY \bullet R \wedge sess \neq null)
\end{aligned}$$

$$
\begin{aligned}
&RHS \\
&\equiv wp(psess = null \wedge Asess = null \rightarrow psess, Asess := ready, sendA, \\
&\qquad\qquad (\exists S \bullet R \wedge \alpha)) \\
&\equiv (psess = null \wedge Asess = null) \Rightarrow \\
&\qquad (\exists sess : STATUS;\ k : KEY \bullet k = pk \wedge sess = ready \wedge \alpha) \\
&\equiv (psess = null \wedge Asess = null) \Rightarrow \alpha[ready, pk/sess, k]
\end{aligned}
$$

Need to show that *RHS* follows for each disjunct of *LHS*. For first disjunct, *RHS* follows immediately. For second disjunct, if $sess \neq null$ then *sess* must be one of $\{ready, sendA, sendB\}$. In each case, R ensures that $\neg\ (psess = null \wedge Asess = null)$ and so *RHS* follows. Therefore: $LHS \equiv\!\!\Rightarrow RHS$.

For *fixkey*:

$$
\begin{aligned}
&LHS \equiv (\exists sess : STATUS;\ k : KEY \bullet R \wedge \\
&\qquad\qquad (sess = ready \Rightarrow (\forall k : KEY \bullet \alpha[sendA/sess]))) \\
&\equiv (\exists sess : STATUS;\ k : KEY \bullet k = pk \wedge sess = ready \wedge \\
&\quad psess = ready \wedge Asess = sendA \wedge (\forall k : KEY \bullet \alpha[sendA/sess])) \vee \\
&\quad (\exists sess : STATUS;\ k : KEY \bullet R \wedge sess \neq ready \\
&\equiv (psess = ready \wedge Asess = sendA \wedge (\forall k : KEY \bullet \alpha[sendA/sess])) \vee \\
&\quad (\exists sess : STATUS;\ k : KEY \bullet R \wedge sess \neq ready)
\end{aligned}
$$

$$
\begin{aligned}
&RHS \equiv wp(psess = ready \rightarrow pk :\in KEY;\ psess := sendA, \\
&\qquad\qquad (\exists sess : STATUS;\ k : KEY \bullet R \wedge \alpha)) \\
&\equiv psess = ready \Rightarrow \\
&\qquad (\forall pk : KEY \bullet \exists sess : STATUS;\ k : KEY \bullet k = pk \wedge \\
&\qquad\qquad sess = sendA \wedge Asess = sendA \wedge \alpha) \\
&\equiv psess = ready \Rightarrow \\
&\qquad (\forall pk : KEY \bullet Asess = sendA \wedge \alpha[sendA, pk/sess, k]) \\
&\equiv psess = ready \Rightarrow (Asess = sendA \wedge (\forall k : KEY \bullet \alpha[sendA/sess]))
\end{aligned}
$$

We show that *RHS* follows from each disjunct of *LHS*. For first disjunct, implication follows immediately. For second disjunct: if $sess \neq ready$ then *sess* must be one of $\{null, sendA, sendB\}$ and R ensures that in each case $psess \neq ready$. Hence *RHS* follows.

For *keytoA* with output value *kv*:

$$
\begin{aligned}
LHS \equiv\ & (\exists\, sess : STATUS;\ k : KEY \bullet R \wedge \\
& \qquad wp(sess = sendA \wedge k = kv \rightarrow sess := sendB, \alpha)) \\
\equiv\ & (\exists\, sess : STATUS;\ k : KEY \bullet R \wedge \\
& \qquad (sess = sendA \wedge k = kv \Rightarrow \alpha[sendB/sess])) \\
\equiv\ & (\exists\, sess : STATUS;\ k : KEY \bullet sess = sendA \wedge k = kv \wedge \\
& \quad k = pk \wedge psess = sendA \wedge Asess = sendA \wedge \alpha[sendB/sess]) \vee \\
& (\exists\, sess : STATUS;\ k : KEY \bullet R \wedge \neg\, (sess = sendA \wedge k = kv)) \\
\equiv\ & (psess = sendA \wedge Asess = sendA \wedge pk = kv \wedge \alpha[sendB, kv/sess, k]) \vee \\
& (\exists\, sess : STATUS;\ k : KEY \bullet R \wedge \neg\, (sess = sendA \wedge k = kv))
\end{aligned}
$$

$$
\begin{aligned}
RHS \equiv\ & wp(psess = sendA \wedge Asess = sendA \wedge pk = kv \rightarrow \\
& \quad Ak, psess, Asess := pk, null, sendB, (\exists\, S \bullet R \wedge \alpha)) \\
\equiv\ & (psess = sendA \wedge Asess = sendA \wedge pk = kv) \Rightarrow \\
& \quad (\exists\, sess : STATUS;\ k : KEY \bullet k = pk \wedge sess = sendB \wedge pk = k \wedge \alpha) \\
\equiv\ & (psess = sendA \wedge Asess = sendA \wedge pk = kv) \Rightarrow \alpha[sendB, pk/sess, k])
\end{aligned}
$$

Again, the *RHS* follows immediately from first disjunct of *LHS*. For the second disjunct of the *LHS*: either $sess \neq sendA$ or $k \neq kv$. In conjunction with R each of these would ensure that the expression to the left of the implication in the *RHS* is false. Hence: $LHS \not\equiv RHS$.

For *keytoB* with output value *kv*:

$$
\begin{aligned}
LHS \equiv\ & (\exists\, sess : STATUS;\ k : KEY \bullet R \wedge \\
& \qquad wp(sess = sendB \wedge k = kv \rightarrow sess := null, \alpha))
\end{aligned}
$$

$$\equiv (\exists\, sess : STATUS;\ k : KEY \bullet k = pk \wedge sess = sendB \wedge$$
$$psess = null \wedge Asess = sendB \wedge Ak = k \wedge k = kv \wedge \alpha[null/sess]) \vee$$
$$(\exists\, sess : STATUS;\ k : KEY \bullet R \wedge \neg\, (sess = sendB \wedge k = kv))$$
$$\equiv (pk = kv \wedge psess = null \wedge Asess = sendB \wedge Ak = kv \wedge$$
$$\alpha[null, kv/sess, k]) \vee$$
$$(\exists\, sess : STATUS;\ k : KEY \bullet R \wedge \neg\, (sess = sendB \wedge k = kv))$$

RHS

$$\equiv wp(Asess = sendB \wedge Ak = kv \rightarrow Asess := null, (\exists\, S \bullet R \wedge \alpha))$$
$$\equiv (Asess = sendB \wedge Ak = kv) \Rightarrow$$
$$(\exists\, sess : STATUS;\ k : KEY \bullet k = pk \wedge sess = null \wedge$$
$$psess = null \wedge \alpha)$$
$$\equiv (Asess = sendB \wedge Ak = kv) \Rightarrow (psess = null \wedge \alpha[null, pk/sess, k])$$

Again, the *RHS* follows from each disjunct of *LHS*. For the second disjunct of the *LHS*: if $R \wedge (sess \neq sendB)$ then $Asess \neq sendB$, so *RHS* follows. Now assume $R \wedge (k \neq kv)$. Either $Asess \neq sendB$, in which case *LHS* follows, or $Asess = sendB$). In the latter case R ensures that $Ak = k$ and so $Ak \neq kv$. This again implies *RHS*.

3. For *startsess*:

$$LHS \equiv (\exists\, sess : STATUS;\ k : KEY \bullet R \wedge sess = null)$$
$$\equiv (\exists\, sess : STATUS;\ k : KEY \bullet$$
$$k = pk \wedge sess = null \wedge Asess = null \wedge psess = null)$$
$$\equiv Asess = null \wedge psess = null$$

For *fixkey*:

$$LHS \equiv (\exists\, sess : STATUS;\ k : KEY \bullet R \wedge sess = ready)$$

$$\equiv (\exists\, sess : STATUS;\ k : KEY \bullet$$
$$k = pk \wedge sess = ready \wedge psess = ready \wedge Asess = sendA)$$
$$\equiv psess = ready \wedge Asess = sendA$$

For *keytoA* with output *kv*:

$$LHS \equiv (\exists\, sess : STATUS;\ k : KEY \bullet k = pk \wedge sess = sendA \wedge$$
$$Asess = sendA \wedge psess = sendA \wedge k = kv)$$
$$\equiv psess = sendA \wedge Asess = sendA \wedge pk = kv$$

For *keytoB* with output *kv*:

$$LHS \equiv (\exists\, sess : STATUS;\ k : KEY \bullet k = pk \wedge sess = sendB \wedge$$
$$Asess = sendB \wedge psess = null \wedge Ak = k \wedge Ak = kv)$$
$$\equiv Asess = sendB \wedge psess = null \wedge Ak = k \wedge Ak = kv$$

In each case, the expression obtained above implies the guard of the corresponding action of the concrete system. Thus all three conditions hold.

Proof of properties for Chapter 8

Proof of security property S2

To show that for all $h : HNAME$, $a : APPID$, $u : USER$, $o : OBJECT$:

$$((h, a, u, granted\ o) \in kernelout1) \Rightarrow$$
$$((currentusers1\ h)\ u \geq (dbase1\ o)) \qquad \textbf{(S2)}$$

by induction over the actions of *SecKer*1. Firstly, it is established by the initialisation since:

$$wp(init, \mathbf{S2})$$

$$\equiv ((h, a, u, \mathit{granted}\ o) \in [\!|\,|\!]) \Rightarrow ((\mathit{currentusers1}\ h)\ u \geq (\mathit{dbase1}\ o))$$
$$\equiv \mathit{true}$$

Suppose **S2** holds, then we need to show that it is preserved by each action, α, of *SecKer*1, that is: $\mathbf{S2} \Rightarrow \mathit{wp}(\alpha, \mathbf{S2})$.

For invoke_h:

The only variable altered by the action is *kernelin*1 so:

$$\mathit{wp}(\mathit{invoke}_h, \mathbf{S2}) \equiv \mathbf{S2}$$

For $\mathit{response}_h$:

For output $(h1, a1, u1, r1)$:

$$\mathit{wp}(\mathit{response}_{(h1,a1,u1,r1)}, \mathbf{S2})$$
$$\equiv ((h1, a1, u1, r1) \in \mathit{kernelout1}) \Rightarrow$$
$$(((h, a, u, \mathit{granted}\ o) \in (\mathit{kernelout1} - [\!|(h1, a1, u1, r1)|\!])) \Rightarrow$$
$$((\mathit{currentusers1}\ h)\ u \geq (\mathit{dbase1}\ o)))$$

which is implied by **S2**.

For *decide*:

$$\mathit{wp}(\mathit{decide}, \mathbf{S2})$$
$$\equiv \mathit{kernelin1} \neq [\!|\,|\!] \Rightarrow$$
$$\mathit{wp}((\mathbf{var}\ (h1, a1, u1, o1) \in \mathit{kernelin1} \bullet \mathit{DECIDE1}), \mathbf{S2})$$
$$\equiv \mathit{kernelin1} \neq [\!|\,|\!] \Rightarrow$$
$$(\forall\ (h1, a1, u1, o1) \in \mathit{kernelin1} \bullet \mathit{wp}(\mathit{DECIDE1}, \mathbf{S2}))$$
$$\equiv \mathit{kernelin1} \neq [\!|\,|\!] \Rightarrow$$
$$(\forall\ (h1, a1, u1, o1) \in \mathit{kernelin1} \bullet$$

$$
\left(
\begin{array}{l}
u! \notin \mathrm{dom}(currentusers1\ h1) \vee \\
a! \notin \mathrm{dom}(regapps1\ h1)
\end{array}
\right) \Rightarrow
$$

$$
\left(
\begin{array}{l}
\mathbf{S2}[kernelin1 - [\!|(h1, a1, u1, o1)|\!]/ \\
\quad kernelin1]
\end{array}
\right)
$$

$$
\wedge
$$

$$
\left(
\begin{array}{l}
u! \in \mathrm{dom}(currentusers1\ h1) \wedge \\
a! \in \mathrm{dom}(regapps1\ h1) \wedge \\
((currentusers1\ h1)\ u1 \notin (regapps1\ h1)\ a1 \vee \\
\ (currentusers1\ h1)\ u1 < dbase1\ o1)
\end{array}
\right) \Rightarrow
$$

$$
\left(
\begin{array}{l}
\mathbf{S2}[kernelin1 - [\!|(h1, a1, u1, o1)|\!], \\
\quad kernelout1 + [\!|(h1, a1, u1, denied\ o1)|\!]/ \\
\qquad kernelin1, kernelout1]
\end{array}
\right)
$$

$$
\wedge
$$

$$
\left(
\begin{array}{l}
u! \in \mathrm{dom}(currentusers1\ h1) \wedge \\
a! \in \mathrm{dom}(regapps1\ h1) \wedge \\
(currentusers1\ h1)\ u1 \in (regapps1\ h1)\ a1 \wedge \\
(currentusers1\ h1)\ u1 \geq dbase1\ o1
\end{array}
\right) \Rightarrow
$$

$$
\left(
\begin{array}{l}
\mathbf{S2}[kernelin1 - [\!|(h1, a1, u1, o1)|\!], \\
\quad kernelout1 + [\!|(h1, a1, u1, granted\ o1)|\!]/ \\
\qquad kernelin1, kernelout1]
\end{array}
\right)
$$

$$
)
$$

To show that $\mathbf{S2} \Rightarrow wp(decide, \mathbf{S2})$. Suppose **S2**, that is:

$$
((h, a, u, granted\ o) \in kernelout1) \Rightarrow
$$

$$
((currentusers1\ h)\ u \geq (dbase1\ o)) \qquad \text{[i]}
$$

suppose also that:

$$
kernelin1 \neq [\!|\,|\!] \qquad \text{[ii]}
$$

and:

$$
(h1, a1, u1, o1) \in kernelin1 \qquad \text{[iii]}
$$

Case 1 If:

$$u! \notin \mathrm{dom}(currentusers1\ h1) \vee a! \notin \mathrm{dom}(regapps1\ h1)$$

then:

$$\begin{aligned}&\mathbf{S2}[kernelin1 - \mathbf{|}(h1, a1, u1, o1)\mathbf{|}]\\ &\equiv ((h, a, u, granted\ o) \in kernelout1) \Rightarrow\\ &\qquad\qquad ((currentusers1\ h)\ u \geq (dbase1\ o))\end{aligned}$$

This is implied by [i].

Case 2 If:

$$\begin{aligned}&u! \in \mathrm{dom}(currentusers1\ h1) \wedge\\ &a! \in \mathrm{dom}(regapps1\ h1) \wedge\\ &((currentusers1\ h1)\ u1 \notin (regapps1\ h1)\ a1 \vee\\ &\ (currentusers1\ h1)\ u1 < dbase1\ o1)\end{aligned}$$

then:

$$\begin{aligned}&\mathbf{S2}[kernelin1 - \mathbf{|}(h1, a1, u1, o1)\mathbf{|},\\ &\quad kernelout1 + \mathbf{|}(h1, a1, u1, denied\ o1)\mathbf{|}/\\ &\qquad kernelin1, kernelout1]\\ &\equiv (h, a, u, granted\ o) \in (kernelout1 + \mathbf{|}(h1, a1, u1, denied\ o1)\mathbf{|}) \Rightarrow\\ &\qquad\qquad (currentusers1\ h)\ u \geq dbase1\ o\end{aligned}$$

Again, this is implied by [i].

Case 3 If:

$$\begin{aligned}&u! \in \mathrm{dom}(currentusers1\ h1) \wedge\\ &a! \in \mathrm{dom}(regapps1\ h1) \wedge\\ &(currentusers1\ h1)\ u1 \in (regapps1\ h1)\ a1 \wedge\\ &(currentusers1\ h1)\ u1 \geq dbase1\ o1 \qquad \text{[iv]}\end{aligned}$$

then:

$$\begin{aligned}&\mathbf{S2}[kernelin1 - \lfloor(h1, a1, u1, o1)\rfloor,\\ &\quad kernelout1 + \lfloor(h1, a1, u1, granted\ o1)\rfloor /\\ &\qquad kernelin1, kernelout1]\\ &\equiv (h, a, u, granted\ o) \in (kernelout1 + \lfloor(h1, a1, u1, granted\ o1)\rfloor) \Rightarrow\\ &\qquad\qquad (currentusers1\ h)\ u \geq dbase1\ o\end{aligned}$$

If $(h, a, u, granted\ o) \in kernelout1$ then this is implied by [i].
If $(h, a, u, granted\ o) = (h1, a1, u1, granted\ o)$ then it is implied by the final conjunct of [iv].

Proof of security property $S4$

Property **S4**:

$$(h, a, u, r) \in kernelout1 \Rightarrow (curentusers1\ h)\ u \geq dbase1\ o$$

This is similar to **S2** and its proof follows the same pattern. Again, *invoke* does not alter $kernelout1$, and *response* only diminishes it, so the property is preserved by these. The $DECIDE1$ definition ensures that an element is only placed in $kernelout1$ if the RHS of **S4** holds.

D.0.5 Proof of deterministic security property

Need to show that:

$$Simple \sqsubseteq obs_H(SecKer1)'$$

with refinement relation:

$$RR \equiv (kernelin0 = kernelin1 \upharpoonright c) \wedge (kernelout0 = kernelout1 \upharpoonright c)$$

The Hiding Refinement Rule (Property 4) can be used to show this. The conditions are verified below, with ϕ representing any condition with no free occurences of concrete variables. The expression:

$$(\exists\, kernelin0, kernelout0 \bullet RR \wedge \phi)$$

occurs below and expands to:

$$\phi[kernelin1 \upharpoonright c, kernelout1 \upharpoonright c / kernelin0, kernelout0]$$

The substitution $kernelin1 \upharpoonright c, kernelout1 \upharpoonright c / kernelin0, kernelout0$ will be denoted $RRrep$.

1. Refinement of initialisation

Required to show:

$$Simple_{init} \preceq'_{RR} obs_H(SecKer1)'_{init}$$

that is:

$$wp(Simple_{init}, \phi) \equiv\!\!\Rightarrow wp(obs_H(SecKer1)'_{init}, \phi[RRrep])$$

$$LHS \equiv wp(kernelin0, kernelout0 := [\,], [\,], \phi)$$
$$\equiv \phi[[\,], [\,] / kernelin0, kernelout0]$$

$$RHS \equiv wp(kernelin1, kernelout1 := [\,], [\,], \phi[RRrep])$$
$$\equiv (\phi[RRrep])[[\,], [\,] / kernelin1, kernelout1]$$
$$\equiv \phi[[\,] \upharpoonright c, [\,] \upharpoonright c / kernelin0, kernelout0]$$
$$\equiv \phi[[\,], [\,] / kernelin0, kernelout0]$$

Therefore $LHS \equiv\!\!\Rightarrow RHS$.

2. Refinement of actions

For each action, α, required to show:

$$Simple_\alpha \preceq_{RR} obs_H(SecKer1)'_\alpha$$

that is:

$$(\exists\, kernelin0, kernelout0 \bullet RR \wedge wp(Simple_\alpha, \phi)) \equiv\!\!\Rightarrow$$
$$wp(obs_H(SecKer1)'_\alpha, (\exists\, kernelin0, kernelout0 \bullet RR \wedge \phi))$$

which is equivalent to:

$$wp(Simple_\alpha, \phi)[RRrep] \equiv\!\!\Rightarrow wp(obs_H(SecKer1)'_\alpha, \phi[RRrep])$$

For *invoke*

Suppose $creq\ r? \leq c$. Then:

$$LHS \equiv (wp(true \rightarrow kernelin0 := kernelin0 + \lfloor r? \rfloor, \phi))[RRrep]$$
$$\equiv \phi[kernelin0 + \lfloor r? \rfloor / kernelin0][RRrep]$$
$$\equiv \phi[(kernelin1 \uparrow c) + \lfloor r? \rfloor, kernelout1 \uparrow c / kernelin0, kernelout0]$$

$$RHS \equiv wp(true \rightarrow kernelin1 := kernelin1 + \lfloor r? \rfloor, \phi[RRrep])$$
$$\equiv (\phi[RRrep])[kernelin1 + \lfloor r? \rfloor / kernelin1]$$
$$\equiv \phi[(kernelin1 + \lfloor r? \rfloor) \uparrow c, kernelout1 \uparrow c / kernelin0, kernelout0]$$
$$\equiv \phi[(kernelin1 \uparrow c) + \lfloor r? \rfloor, kernelout1 \uparrow c / kernelin0, kernelout0]$$
[Since $creq\ r? \leq c$]

Therefore $LHS \equiv\!\!\Rightarrow RHS$.

Suppose $creq\ r? > c$. Then:

$$LHS \equiv wp(skip, \phi)[RRrep] \equiv \phi[RRrep]$$

$$RHS \equiv wp((true \rightarrow kernelin1 := kernelin1 + \lfloor r? \rfloor) \,[]\, skip, \phi[RRrep])$$
$$\equiv \phi[(kernelin1 + \lfloor r? \rfloor) \uparrow c, kernelout1 \uparrow c / kernelin0, kernelout0] \wedge$$
$$\phi[RRrep]$$
$$\equiv \phi[kernelin1 \uparrow c, kernelout1 \uparrow c / kernelin0, kernelout0]$$
[Since $creq\ r? > c$]
$$\equiv \phi[RRrep]$$

Therefore $LHS \equiv\!\!\Rightarrow RHS$.

For *decide*

$$LHS \equiv (wp(kernelin0 \neq [\!] \rightarrow$$
$$(\mathbf{var}\ m \in kernelin0 \bullet kernelin0, kernelout0 :=$$

$$kernelin0 - \lfloor m \rfloor, kernelout0 + (f\ m), \phi)))[RRrep]$$

$$\equiv (kernelin0 \neq [\,] \Rightarrow (\forall m \in kernelin0 \bullet$$

$$\phi[kernelin0 - \lfloor m \rfloor, kernelout0 + (f\ m)/$$

$$kernelin0, kernelout0]))[RRrep]$$

$$\equiv kernelin1 \upharpoonright c \neq [\,] \Rightarrow (\forall m \in kernelin1 \upharpoonright c) \bullet$$

$$\phi[(kernelin1 \upharpoonright c) - \lfloor m \rfloor, (kernelout1 \upharpoonright c) + (f\ m)/$$

$$kernelin0, kernelout0])$$

$$RHS \equiv wp(kernelin1 \upharpoonright c \neq [\,] \rightarrow DECIDE1, \phi[RRrep])$$

$$\equiv kernelin1 \upharpoonright c \neq [\,] \Rightarrow (\forall m \in kernelin1 \upharpoonright c) \bullet$$

$$(\phi[RRrep])[kernelin1 - \lfloor m \rfloor, kernelout1 + (f\ m)/$$

$$kernelin1, kernelout1])$$

$$\equiv kernelin1 \upharpoonright c \neq [\,] \Rightarrow (\forall m \in kernelin1 \upharpoonright c) \bullet$$

$$\phi[(kernelin1 - \lfloor m \rfloor) \upharpoonright c, (kernelout1 + (f\ m) \upharpoonright c/$$

$$kernelin0, kernelout0])$$

$$\equiv kernelin1 \upharpoonright c \neq [\,] \Rightarrow (\forall m \in kernelin1 \upharpoonright c) \bullet$$

$$\phi[(kernelin1 \upharpoonright c) - \lfloor m \rfloor, (kernelout1 \upharpoonright c) + (f\ m))/$$

$$kernelin0, kernelout0])\,[\text{Since } m \in kernelin1 \upharpoonright c]$$

Therefore $LHS \equiv\!\!\!\!\!\!\Rightarrow RHS$.

For *response*

Suppose $cresp\ r! \leq c$:

LHS

$$\equiv (wp(r! \in kernelout0 \rightarrow kernelout0 := kernelout0 - \lfloor r! \rfloor, \phi))[RRrep]$$

$$\equiv r! \in kernelin1 \upharpoonright c \Rightarrow$$

$$\phi[kernelin1 \upharpoonright c, (kernelout1 \upharpoonright c) - \lfloor r! \rfloor / kernelin0, kernelout0]$$

RHS

$$\equiv wp(r! \in kernelout1 \rightarrow kernelout1 := kernelout1 - \lfloor r! \rfloor, \phi[RRrep])$$
$$\equiv r! \in kernelout1 \Rightarrow$$
$$\quad \phi[kernelin1 \uparrow c, (kernelout1 - \lfloor r! \rfloor) \uparrow c/kernelin0, kernelout0]$$

Again, using the assumption that $cresp\ r! \leq c$, it follows that $LHS \equiv\!\!\!\Rightarrow RHS$.

Suppose $cresp\ r! > c$:

$$LHS \equiv wp(skip, \phi)[RRrep] \equiv \phi[RRrep]$$

$$RHS \equiv wp((r! \in kernelout1 \rightarrow kernelout1 := kernelout1 - \lfloor r! \rfloor)$$
$$\quad [\!] skip, \phi[RRrep])$$
$$\equiv (r! \in kernelout1 \Rightarrow \phi[kernelin1 \uparrow c, (kernelout1 - \lfloor r! \rfloor) \uparrow c/$$
$$\quad kernelin0, kernelout0]) \wedge \phi[RRrep]$$
$$\equiv (r! \in kernelout1 \Rightarrow \qquad \text{[Since } cresp\ r! > c\text{]}$$
$$\quad \phi[kernelin1 \uparrow c, kernelout1 \uparrow c/kernelin0, kernelout0]) \wedge \phi[RRrep]$$
$$\equiv \phi[RRrep]$$

Therefore $RHS \equiv\!\!\!\Rightarrow LHS$ as required.

3. Concrete internal actions

This condition checks that all internal actions introduced at the concrete level refine *skip*. In this case we need to show that:

$$wp(skip, \phi)[RRrep] \equiv\!\!\!\Rightarrow wp(decidehigh, \phi[RRrep])$$

$$LHS \equiv \phi[RRrep]$$

$$RHS$$
$$\equiv wp(kernelin1 - (kernelin1 \uparrow c) \neq \{\!|\} \rightarrow$$
$$\quad (\mathbf{var}\ m \in (kernelin1 - (kernelin1 \uparrow c)) \bullet DECIDE1, \phi[RRrep])$$

$$\equiv kernelin1 - (kernelin1 \upharpoonright c) \neq [\!] \Rightarrow$$
$$(\forall m \in kernelin1 - (kernelin1 \upharpoonright c) \bullet$$
$$\phi[(kernelin1 - [\![m]\!]) \upharpoonright c, (kernelout1 + (f\ m)) \upharpoonright c /$$
$$kernelin0, kernelout0])$$
$$\equiv kernelin1 - (kernelin1 \upharpoonright c) \neq [\!] \Rightarrow$$
$$(\forall m \in kernelin1 - (kernelin1 \upharpoonright c) \bullet$$
$$\phi[kernelin1 \upharpoonright c, kernelout1 \upharpoonright c / kernelin0, kernelout0])$$
[Since $creq\ m > c$ and hence also $cresp\ (f\ m) > c$]
$$\equiv kernelin1 - (kernelin1 \upharpoonright c) \neq [\!] \Rightarrow$$
$$(\forall m \in kernelin1 - (kernelin1 \upharpoonright c) \bullet \phi[RRrep])$$

Hence $LHS \equiv\!\!\rangle\ RHS$.

4. Variant

The variant is provided by the number of high level message in the input bag: $\sharp(kernelin1 - (kernelin1 \upharpoonright c))$. Values of this expression are members of $\mathbb{N}$ which is well-founded.

5. Decreasing the variant

Each execution of the internal action *decidehigh* removes one element from the set $(kernelin1 - (kernelin1 \upharpoonright c))$. So the variant is decreased each time. Hence execution of internal actions is bounded.

6. Refinement of guards

For action, α, need to show:

$$gd(Simple_\alpha)[RRrep] \equiv\!\!\rangle$$
$$(gd(obs_H(SecKer)'_\alpha) \vee gd(obs_H(SecKer)'_{decidehigh}))$$

For *invoke*

Both abstract and concrete actions have guard *true*, so the condition holds for *invoke*.

For *decide*

$$LHS \equiv kernelin1 \upharpoonright c \neq \mathbb{I}$$

$$RHS \equiv (kernelin1 \upharpoonright c \neq \mathbb{I}) \vee (kernelin1 - (kernelin1 \upharpoonright c) \neq \mathbb{I})$$

So again $LHS \rightrightarrows RHS$.

For *response*

$$LHS \equiv (cresp\ r! \leq c \wedge r! \in kernelin1 \upharpoonright c) \vee (cresp\ r! > c)$$

$$RHS \equiv (cresp\ r! \leq c \wedge r! \in kernelin1) \vee (cresp\ r! > c) \vee (kernelin1 - (kernelin1 \upharpoonright c) \neq \mathbb{I})$$

So the property also holds for *response*.

Each of the conditions has been verified, proving the required refinement.

Proof of second refinement

We need to show that:

$$SecKer2 = (\|\ n \in HNAME \bullet SecKer_n) \setminus \{m, n : HNAME \bullet transfer_{(m,n)}\}$$

First, the parallel composition of all the $SecKer_n$ action systems is calculated by repeated application of the composition rule from Section 4.4. The combined state is formed by the union of the $Host2_n$:

$$(\cup\ n \in HNAME \bullet Hosts2_n)$$
$$\equiv (\cup\ n \in HNAME \bullet n \mapsto host2\ n) \qquad \text{[Equivalence from Ch. 8]}$$
$$\equiv Hosts2$$

get_n

guard
$(hosts2\ n).kernelin2 \neq [\!]$

command
(**var** $a : APPID;\ u : USER;\ o : OBJECT \mid$
$(a, u, o) \in (hosts2\ n).kernelin2 \bullet DECIDE2$)

Figure D.2 The indexed internal action get_n

In fact, the state is partitioned between the concrete action systems. The initialisations are combined as follows:

$$(||\ n \in HNAME \bullet (kernelin2_n, kernelout2_n, pending_n := [\!], [\!], [\!])$$
$$\equiv (||\ n \in HNAME \bullet$$
$$(hosts2\ n).kernelin2, (hosts2\ n).kernelout2, (hosts2\ n).pending :=$$
$$[\!], [\!], [\!])$$
$$\equiv SecKer2_{init}$$

For actions $invoke_n$ and $response_n$ the labels are unique, so no labelled actions have to be composed. The identity above ensures that the combined actions are equivalent to the corresponding actions from $SecKer2$.

An internal action which selects an arbitrary value is equivalent to an indexed set of internal actions (proof in [20]), that is:

internal $h\ : -\ u : [(\exists\, x \in X \bullet post_x]$

is equivalent to:

for $x \in X$ **internal** $h\ : -\ u : [post_x]$

This means that, for example, the internal action *getrequest* is equivalent to:

for $n \in HNAME \bullet$ **internal** get_n

with get_n as defined in Figure D.2. Using the identity on state components,

each indexed action is equivalent to its counterpart from $SecKer_n$ as required. The results for *decide* and *deliver* follow in a similar manner.

The only actions which will need to be combined are the *transfer* actions. In transferring a pending item from host m to host n the two actions to be combined are, from $SecKer_m$:

$$\textbf{action } transfer_{(m,n)} \textbf{ out } p! : PENDING \ : -$$
$$n \neq m \wedge (p! \in pending_m) \wedge (destination\ p! = n) \rightarrow$$
$$pending_m := pending_m - \lfloor p! \rfloor$$

and from $SecKer_n$:

$$\textbf{action } transfer_{(m,n)} \textbf{ in } p? : PENDING \ : -$$
$$true \rightarrow pending_n := pending_n + \lfloor p? \rfloor$$

Using the parallel composition rule for value passing action systems (Section 4.4) the composition of these two is:

$$\textbf{action } transfer_{(m,n)} \textbf{ out } p! : PENDING \ : -$$
$$n \neq m \wedge (p! \in pending_m) \wedge (destination\ p! = n) \rightarrow$$
$$pending_m, pending_n := pending_m - \lfloor p! \rfloor, pending_n + \lfloor p! \rfloor$$

The conditions required for this are ensured since the original input action is always willing to accept any input of type *PENDING*.

All these actions for each m, n are to be hidden. First, the output variable $p!$ is localised. This gives:

$$\textbf{action } transfer_{(m,n)} \ : -$$
$$(\exists\, p : PENDING \bullet n \neq m \wedge (p \in pending_m) \wedge$$
$$(destination\ p = n)) \rightarrow$$
$$(\textbf{var } p : PENDING \bullet pending_m, pending_n :=$$
$$pending_m - \lfloor p \rfloor, pending_n + \lfloor p \rfloor)$$

When internalised these actions can be combined as before to give:

$$
\begin{array}{l}
\textbf{action } \textit{transfer} \ : - \\
\quad (\exists\, m, n : \mathit{HNAME};\ p : \mathit{PENDING} \bullet \\
\qquad\qquad n \neq m \wedge (p \in \mathit{pending}_m) \wedge (\mathit{destination}\ p = n)) \rightarrow \\
\qquad (\textbf{var } m, n : \mathit{HNAME};\ p : \mathit{PENDING} \bullet \mathit{pending}_m, \mathit{pending}_n := \\
\qquad\qquad\qquad \mathit{pending}_m - \lfloor p \rfloor, \mathit{pending}_n + \lfloor p \rfloor)
\end{array}
$$

Using the One Point Rule this is equivalent to:

$$
\begin{array}{l}
\textbf{action } \textit{transfer} \ : - \\
\quad (\exists\, m : \mathit{HNAME};\ p : \mathit{PENDING} \bullet \\
\qquad\qquad (\mathit{destination}\ p \neq m) \wedge (p \in \mathit{pending}_m)) \rightarrow \\
\qquad (\textbf{var } m : \mathit{HNAME};\ p : \mathit{PENDING} \bullet \\
\qquad\qquad \mathit{pending}_m, \mathit{pending}_{(\mathit{destination}\ p)} := \\
\qquad\qquad\qquad \mathit{pending}_m - \lfloor p \rfloor, \mathit{pending}_{(\mathit{destination}\ p)} + \lfloor p \rfloor)
\end{array}
$$

and using the identity for state between the two levels this in turn is equivalent to:

$$
\begin{array}{l}
\textbf{action } \textit{transfer} \ : - \\
\quad (\exists\, m : \mathit{HNAME};\ p : \mathit{PENDING} \bullet \\
\qquad\qquad (\mathit{destination}\ p \neq m) \wedge (p \in (\mathit{host2}\ m).\mathit{pending})) \rightarrow \\
\qquad (\textbf{var } m : \mathit{HNAME};\ p : \mathit{PENDING} \bullet \\
\qquad\qquad (\mathit{host2}\ m).\mathit{pending}, (\mathit{host2}\ (\mathit{destination}\ p)).\mathit{pending} := \\
\qquad\qquad\qquad (\mathit{host2}\ m).\mathit{pending} - \lfloor p \rfloor, \\
\qquad\qquad\qquad (\mathit{host2}\ (\mathit{destination}\ p)).\mathit{pending} + \lfloor p \rfloor)
\end{array}
$$

which is the definition of *transfer* in $SecKer2$. Hence the parallel composition of the $SecKer_n$ with the $transfer_{(m,n)}$ actions hidden is equivalent to $SecKer2$ as required.

Bibliography

[1] M. Abadi and L. Lamport. The existence of refinement mappings. In *Proceedings of the 3rd IEEE Symposium on Logic in Computer Science*, pages 165–177. IEEE Computer Society Press, 1988.

[2] J.-R. Abrial. *The B-Book.* Cambridge University Press, 1996.

[3] P.G. Allen. A comparison of non-interference and non-deducibility using CSP. In *Proceedings of the 4th IEEE Computer Security Foundations Workshop*, pages 43–54. IEEE Computer Society Press, 1991.

[4] Edward Amoroso. *Fundamentals of Computer Security Technology.* Prentice Hall, 1994.

[5] R.D. Arthan. A method for specifying secure systems. Technical Report DS/FMU/RDA/13, ICL Defence Systems Formal Methods Unit, Eskdale Road, Winnersh, Berks, UK, RG11 5TT, 1989.

[6] R.J.R. Back. Refinement calculus II: Parallel and reactive programs. In W-P. de Roever J.W. de Bakker and G. Rozenburg, editors, *Proc REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, LNCS 430, pages 67–93. Springer-Verlag, 1990.

[7] R.J.R. Back and R. Kurki-Suonio. Decentralisation of process nets with centralised control. In *2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 131–142, 1983.

[8] R.J.R. Back and K. Sere. Refinement of action systems. In *Mathematics of Program Construction*, LNCS 375. Springer-Verlag, 1989.

[9] R.J.R. Back and K. Sere. Deriving an Occam implementation of action systems. In Carroll Morgan and J.C.P. Woodcock, editors, *Proceedings of 3rd BCS-FACS Refinement Workshop*, LNCS 430, pages 9–30. Springer-Verlag, 1990.

[10] R.J.R. Back and J. von Wright. Refinement calculus I: Sequential nondeterministic programs. In W-P. de Roever J.W. de Bakker and G. Rozenburg, editors, *Proc REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, LNCS 430, pages 42–66. Springer-Verlag, 1990.

[11] D.E. Bell and L.J. LaPadula. Secure computer systems: Mathematical foundations and model. Technical Report M74-244, The MITRE Corporation, Bedford, MA, 01730, 1974.

[12] D.E. Bell and L.J. LaPadula. Secure computer system: Unified exposition and multics interpretation. Technical Report ESD-TR-75-306, The MITRE Corporation, Bedford, MA, 01730, March 1976.

[13] William R. Bevier and William D. Young. A state-based approach to noninterference. In *Proceedings of the 7th IEEE Computer Security Foundations Workshop*, pages 11–21. IEEE Computer Society Press, 1994.

[14] P. Bieber, N. Boulahia-Cuppens, T. Lehmann, and E. van Wickeren. Abstract machines for communications security. In *Proceedings of the 6th IEEE Computer Security Foundations Workshop.* IEEE Computer Society Press, 1993.

[15] Pierre Bieber and Frédéric Cuppens. A logical view of secure dependencies. *Journal of Computer Security*, 1(1):99–129, 1992.

[16] Anthony Boswell. Specification and validation of a security policy model. *IEEE Transactions on Software Engineering*, 21(2), February 1995.

[17] R. Burnham. The specification of security in distributed computing systems. Master's thesis, Oxford University, 1987.

[18] M. Butler. Stepwise refinement of communicating systems. Technical Report Ser. A. No 147, Abo Akademi University, 1994.

[19] M. Butler. Action systems and security protocols. Draft, 1997.

[20] M.J. Butler. *A CSP Approach to Action Systems.* PhD thesis, Oxford University, 1992.

[21] M.J. Butler et al. Specification of a program derivation editor. Reports in Mathematics and Computer Science A94-157, Abo Akademi University, Finland, 1997.

[22] M.J. Butler and C.C. Morgan. Action systems, unbounded nondeterminism and infinite traces. *Formal Aspects of Computing*, 7:37–53, 1994.

[23] Ana Cavalcanti and Jim Woodcock. A weakest precondition semantics for Z. Technical Report TR-24-95, Programming Research Group, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, OX1 3QD, November 1995.

[24] D.D. Clark and D.R. Wilson. A comparison of commercial and military computer security policies. In *Proceedings of the 8th IEEE Symposium on Security and Privacy.* IEEE Computer Society Press, 1987.

[25] Rodger Collinson. Proving critical properties of functional specifications. Technical report, Communications and Electronic Security Group, 1993. Draft 0.2.

[26] D.E. Comer. *Internetworking with TCP/IP: Principles, Protocols and Architecture.* Prentice Hall, 1988.

[27] F. Cuppens. A logical formalization of secrecy. In *Proceedings of the 6th IEEE Computer Security Foundations Workshop.* IEEE Computer Society Press, 1993.

[28] D. E. Denning. *Secure information flow in computer systems.* PhD thesis, Purdue University, 1975.

[29] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[30] P. J. Denning. Third generation computer systems. *Computing Surveys*, 3(4):171–216, December 1971.

[31] E.W Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[32] G. W. Dinolt, L. A. Benzinger, and M. G. Yatabe. Combining components and policies. In *Proceedings of the 7th IEEE Computer Security Foundations Workshop*, pages 22–33. IEEE Computer Society Press, 1994.

[33] Todd Fine. A foundation for covert channel analysis. In *Proceedings of the 15th National Computer Security Conference*, pages 204–212, Baltimore, USA, 1992.

[34] Todd Fine, J. Thomas Haigh, Richard O'Brien, and Dana L. Toups. Noninterference and unwinding for lock. In *Proceedings of the 2nd IEEE Computer Security Foundations Workshop.* IEEE Computer Society Press, 1989.

[35] Riccardo Focardi and Roberto Gorrieri. A taxonomy of trace-based security properties for CCS. In *Proceedings of the 7th IEEE Computer Security Foundations Workshop.* IEEE Computer Society Press, 1994.

[36] Simon N. Foley. A universal theory of information flow. In *Proceedings of the 8th IEEE Symposium on Security and Privacy*, pages 116–122, Oakland, CA., 1987. IEEE Computer Society Press.

[37] Simon N. Foley. A model for secure information flow. In *Proceedings of the 10th IEEE Symposium on Security and Privacy*, pages 248–258, Oakland, CA., 1989. IEEE Computer Society Press.

[38] Simon N. Foley. Secure information flow using security groups. In *Proceedings of the 3rd IEEE Computer Security Foundations Workshop*, pages 62–72. IEEE Computer Society Press, 1990.

[39] Simon N. Foley. Aggregation and separation as noninterference properties. *Journal of Computer Security*, 1(2):159–188, 1992.

[40] Simon N. Foley. Reasoning about confidentiality requirements. In *Proceedings of the 7th IEEE Computer Security Foundations Workshop*, pages 150–160. IEEE Computer Society Press, 1994.

[41] Warwick Ford. *Computer Communications Security.* Prentice Hall, 1994.

[42] Janice Glasgow, Glenn MacEwen, and Prakash Panangaden. A logic for reasoning about security. In *Proceedings of the 3rd IEEE Computer Security Foundations Workshop.* IEEE Computer Society Press, 1990.

[43] J.A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the 3rd IEEE Symposium on Security and Privacy*, pages 11–20, Oakland, CA., 1982. IEEE Computer Society Press.

[44] Joseph A. Goguen and José Meseguer. Unwinding and inference control. In *Proceedings of the 5th IEEE Symposium on Security and Privacy*, pages 75–86, Oakland, CA., 1984. IEEE Computer Society Press.

[45] M.J. Gordon. HOL: a machine oriented formulation of higher order logic. Technical Report 68, University of Cambridge Computer Laboratory, 1985.

[46] G.S. Graham and P.J. Denning. Protection: principles and practice. In *Proceedings of the AFIPS Spring Joint Computer Conference*, volume 40, pages 417–429. AFIPS Press, 1972.

[47] John Graham-Cumming. *The Formal Development of Secure Systems.* PhD thesis, Oxford University, 1992.

[48] John Graham-Cumming. Some laws of non-interference. In *Proceedings of the 5th IEEE Computer Security Foundations Workshop*, pages 22–33. IEEE Computer Society Press, 1992.

[49] John Graham-Cumming. Laws of non-interference in CSP. *Journal of Computer Security*, 2(1):37–52, 1993.

[50] James W. Gray III. Probabilistic interference. In *Proceedings of the 11th IEEE Symposium on Security and Privacy*, pages 170–179, Oakland, CA., 1990. IEEE Computer Society Press.

[51] James W. Gray III. Toward a mathematical foundation for information flow security. In *Proceedings of the 12th IEEE Symposium on Security and Privacy*, pages 21–34, Oakland, CA., 1991. IEEE Computer Society Press.

[52] James W. Gray III. Towards a mathematical foundation for information flow security. *Journal of Computer Security*, 1(3):255–294, 1992.

[53] James W. Gray III and Paul F. Syverson. A logical approach to multilevel security of probabilistic systems. In *Proceedings of the 13th IEEE Symposium on Security and Privacy*, pages 164–176, Oakland, CA., 1992. IEEE Computer Society Press.

[54] David Guaspari, Mike Seager, and Matt Stillerman. Specifying the kernel of a secure distributed operating system. In Michael G. Hinchey and Jonthan P. Bowen, editors, *Applications of formal methods*, chapter 12, pages 285–437. Prentice Hall, 1995.

[55] J. V. Guttag and J.J. Horning. *Larch:Languages and Tools for Formal Specification.* Texts and Monographs in Computer Science. Springer Verlag, New York, 1993.

[56] Joshua D. Guttman and Mark E. Nadel. What needs securing? In *Proceedings of the 1st Computer Security Foundations Workshop*, pages 34–57, Bedford, MA, 01730, June 1988. The MITRE Corporation.

[57] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, August 1976.

[58] J. He, C.A.R. Hoare, and J.W. Sanders. Data refinement refined. In *European Symposium on Programming*, LNCS 213. Springer-Verlag, 1986.

[59] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[60] ISO/IEC 7498-1. *Information Technology - Open Systems Interconnection - Basic Refence Model.*

[61] Jeremy Jacob. A security framework. In *Proceedings of the 1st Computer Security Foundations Workshop*, pages 98–111, Bedford, MA, 01730, June 1988. The MITRE Corporation.

[62] Jeremy Jacob. Security specifications. In *Proceedings of the 9th IEEE Symposium on Security and Privacy*, pages 14–23, Oakland, CA., 1988. IEEE Computer Society Press.

[63] Jeremy Jacob. Categorising non-interference. In *Proceedings of the 3rd IEEE Computer Security Foundations Workshop*, pages 44–50, 1990.

[64] Jeremy Jacob. Separability and the detection of hidden channels. *Information Processing Letters*, 34(1):27–29, 1990.

[65] Jeremy Jacob. The varieties of refinement. In *Proceedings of the 4th BCS-FACS Refinement Workshop*, LNCS, pages 441–455. Springer-Verlag, 1991.

[66] Jeremy Jacob. Basic theorems about security. *Journal of Computer Security*, 1(4):385–411, 1992.

[67] R. Jain and C. Landwehr. On access checking in capability-based systems. *IEEE Transactions on Software Engineering*, 13(2), 1987.

[68] Dale M. Johnson and F. Javier Thayer. Security and the composition of machines. In *Proceedings of the 1st Computer Security Foundations Workshop*, pages 72–89, Bedford, MA, 01730, June 1988. The MITRE Corporation.

[69] Dale M. Johnson and F. Javier Thayer. Security properties consistent with the testing semantics for communicating processes. In *Proceedings of the 2nd Computer Security Foundations Workshop*, pages 9–21. IEEE Computer Society Press, 1989.

[70] Charlie Kaufman, Radia Perlman, and Mike Speciner. *Network Security.* Prentice Hall, 1995.

[71] Richard A. Kemmerer. Shared resource matrix methodology: an approach to identifying storage and timing channels. *ACM Transactions on Computer Systems*, 1:256–277, August 1983.

[72] Imre Lakatos. *Proofs and Refutations: the logic of mathematical discovery.* Cambridge University Press, 1979.

[73] L. Lamport. A simple approach to specifying concurrent systems. *Communications of the ACM*, 32:32–45, 1989.

[74] B. W. Lampson. Protection. In *Proceedings of the 5th Princeton Symposium on Information Sciences and Systems*, pages 437–443, 1971. Reprinted in ACM SIGOPS Operating Systems Review 8(1):18– 24, Jan. 1974.

[75] B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16:613–615, 1973.

[76] J. Landauer and T. Redmond. A framework for composition of security models. In *Proceedings of the 5th Computer Security Foundations Workshop*, pages 157–165. IEEE Computer Society Press, 1992.

[77] Carl E. Landwehr. Formal models for computer security. *Computing Surveys*, 13:247–278, September 1981.

[78] Gavin Lowe. Some new attacks upon security protocols. In *Proceedings of the 9th Computer Security Foundations Workshop*, pages 162–169. IEEE Computer Society Press, 1996.

[79] Gavin Lowe. Casper: a compiler for the analysis of security protocols. In *Proceedings of the 10th Computer Security Foundations Workshop*, pages 18–30. IEEE Computer Society Press, 1997.

[80] Daryl McCullough. Specifications for multi-level security and a hook-up property. In *Proceedings of the 8th IEEE Symposium on Security and Privacy*, pages 161–166, Oakland, CA., 1987. IEEE Computer Society Press.

[81] Daryl McCullough. Noninterference and the composability of security properties. In *Proceedings of the 9th IEEE Symposium on Security and Privacy*, pages 177–186, Oakland, CA., 1988. IEEE Computer Society Press.

[82] Daryl McCullough. A hookup theorem for multilevel security. *IEEE Transactions on Software Engineering*, 16(6):563–568, June 1990. IEEE Computer Society Press.

[83] J. McLean. A comment on the basic security theorem of Bell and LaPadula. *Information Processing Letters*, 20:67–70, 1985.

[84] J. McLean and C. Meadows. Composable security properties. *Cipher*, pages 27–37, 1989.

[85] John McLean. A formal method for the abstract specification of software. *Journal of the ACM*, pages 600–627, July 1984.

[86] John McLean. Reasoning about security models. In *Proceedings of the 8th IEEE Symposium on Research in Security and Privacy*, pages 123–131, Oakland, CA., 1987. IEEE Computer Society Press.

[87] John McLean. Proving noninterference and functional correctness using traces. *Journal of Computer Security*, 1(1):37–57, 1992.

[88] John McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proceedings of the 15th IEEE Symposium on Research in Security and Privacy*, pages 79–93, Oakland, CA., 1994. IEEE Computer Society Press.

[89] Jonathan Millen. Security kernel validation in practice. *Communications of the ACM*, 19(5):243–250, May 1976.

[90] Jonathan Millen. Foundations of covert channel detection. Technical Report MTR10538, The MITRE Corporation, Bedford, MA, 01730, January 1989.

[91] Jonathan Millen. Hookup security for synchronous machines. In *Proceedings of the 3rd IEEE Computer Security Foundations Workshop*, pages 84–90. IEEE Computer Society Press, 1990.

[92] Jonathan Millen. Unwinding forward correctability. In *Proceedings of the 7th IEEE Computer Security Foundations Workshop*, pages 2–10. IEEE Computer Society Press, 1994.

[93] C.C. Morgan. Of wp and CSP. In D. Gries W.H.J. Feijen, A.G.M. van Gasteren and J. Misra, editors, *Beauty is our business: a birthday salute to Edsger W. Dijkstra.* Springer-Verlag, 1990.

[94] C.C. Morgan. *Programing from Specifications.* International Series in Computer Science. Prentice Hall, 2 edition, 1994.

[95] C.C. Morgan, K.A. Robinson, and P.H.B. Gardiner. On the refinement calculus. Technical monograph PRG-70, Programming Research Group, Oxford University, 1988.

[96] J.M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Sci. Comp. Prog.*, 9:298–306, 1987.

[97] Ira S. Moskowitz and Oliver L. Costich. A classical automata approach to noninterference type problems. In *Proceedings of the 5th IEEE Computer Security Foundations Workshop*, pages 2–8. IEEE Computer Society Press, 1992.

[98] Colin O'Halloran. A calculus of information flow. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, Lecture Notes in Computer Science, pages 147–159. Springer-Verlag, 1990.

[99] Colin O'Halloran. On requirements and security in a CCIS. In *Proceedings of the 5th IEEE Computer Security Foundations Workshop*, pages 121–134. IEEE Computer Society Press, 1992.

[100] Greg O'Shea. On the specification, validation and verification of security in access control systems. *The Computer Journal*, 37(5), 1994.

[101] Ramesh V. Peri, William A. Wulf, and Darrell M. Kienzle. A logic of composition for information flow predicates. In *Proceedings of the 9th IEEE Computer Security Foundations Workshop*, pages 82–93. IEEE Computer Society Press, 1996.

[102] Charles P. Pfleeger. *Security in Computing.* Prentice Hall, 1989.

[103] Sylvan Pinsky. An algebraic approach to non-interference. In *Proceedings of the 5th IEEE Computer Security Foundations Workshop*, pages 34–47. IEEE Computer Society Press, 1992.

[104] Ben Potter, Jane Sinclair, and David Till. *An Introduction to Formal Specification and Z.* Prentice Hall, 1991.

[105] A.W. Roscoe. Model checking CSP. In A.W. Roscoe, editor, *A Classical Mind.* Prentice Hall, 1994.

[106] A.W. Roscoe. Prospects for describing, specifying and verifying key-exchange protocols in CSP and FDR. Technical report, Formal Systems, December 1994.

[107] A.W. Roscoe. CSP and determinism in security modelling. In *Proceedings of the 16th IEEE Symposium on Security and Privacy*, pages 114–127. IEEE Computer Society Press, 1995.

[108] A.W. Roscoe. Modelling and verifying key-exchange protocols using CSP and FDR. In *Proceedings of the 8th IEEE Computer Security Foundations Workshop*, pages 98–107. IEEE Computer Society Press, 1995.

[109] A.W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1997.

[110] A.W. Roscoe, J.C.P. Woodcock, and L. Wulf. Non-interference through determinism. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, Lecture Notes in Computer Science 875, pages 33–53. Springer-Verlag, 1994.

[111] A.W. Roscoe and L. Wulf. Composing and decomposing systems under security properties. In *Proceedings of the 8th IEEE Computer Security Foundations Workshop*, pages 9–15. IEEE Computer Society Press, 1995.

[112] John Rushby. Design and verification of secure systems. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, pages 12–21, Pacific Grove, CA., USA, December 1981.

[113] John Rushby. Proof of separability: A verification technique for a class of security kernels. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA 94025, USA, 1982.

[114] John Rushby. The SRI security model. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA 94025, USA, 1985.

[115] Deborah Russell and G.T.Gangemi Sr. *Computer Security Basics.* O'Reilly & Assoc. Inc., 1989.

[116] P.Y.A. Ryan. A CSP formulation of non-interference. *Cipher*, pages 19–27, 1991. IEEE Computer Society Press.

[117] Ravi S. Sandhu. A lattice interpretation of the Chinese Wall policy. In *Proceedings of the 15th NIST-NCSC National Computer Security Conference*, pages 329–339, 1992.

[118] Ravi S. Sandhu. Lattice-based access control models. *IEEE Computer*, pages 9–19, 1993. IEEE Computer Society Press.

[119] S.A. Schneider. Security properties and CSP. In *Proceedings of the 17th IEEE Symposium on Security and Privacy.* IEEE Computer Society Press, 1996.

[120] S.A. Schneider. Using CSP for protocol analysis: the Needham-Schroeder public-key protocol. Technical report CSD-TR-96-14, Royal Holloway, University of London, 1996.

[121] Mike Seager, David Guaspari, Matt Stillerman, and Carla Marceau. Formal methods in the theta kernel. In *Proceedings of the 16th IEEE Symposium on Security and Privacy*, pages 88–100. IEEE Computer Society Press, 1995.

[122] Jane Sinclair and Darrel Ince. The use of Z in specifying security properties. In H. Habrias, editor, *7th International Conference on putting into practice methods and tools for information system design, ISBN: 2-906082-19-8*, pages 27–39, IRIN, Universite de Nantes, France, October 1995.

[123] Jane Sinclair and Jim Woodcock. Event refinement in state-based concurrent systems. *Formal Aspects of Computing*, 7:266–288, 1995.

[124] E.H. Spafford. The Internet worm: Crisis and aftermath. *Communications of the ACM*, 32(6), 1989.

[125] J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, 2nd edition, 1992.

[126] Clifford Stoll. *The Cuckoo's Egg.* Doubleday, 1989.

[127] David Sutherland. A model of information. In *Proceedings of the 9th National Computer Security Conference*, pages 175–183. U. S. National Computer Security Center and U. S. National Bureau of Standards, 1986.

[128] Ian Sutherland, Stanley Perlo, and Rammohan Varadarajan. Deducibility security with dynamic level assignments. In *Proceedings of the 2nd IEEE Computer Security Foundations Workshop*, pages 3–8. IEEE Computer Society Press, 1989.

[129] A. Tarski. A lattice-theoretic fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5, 1955.

[130] M. Walden and K. Sere. Refining action systems within B-tool. In *Proceedings of FME 96*. IEEE Computer Society Press, 1996.

[131] J. Todd Wittbold and Dale M. Johnson. Information flow in nondeterministic systems. In *Proceedings of the 11th IEEE Symposium on Research in Security and Privacy*, pages 144–161, Oakland, CA., 1990. IEEE Computer Society Press.

[132] J.C.P. Woodcock and C.C. Morgan. Refinement of state-based concurrent systems. In *Proceedings of the VDM Symposium*, LNCS 428. Springer-Verlag, 1990.

[133] Jim Woodcock and Jim Davies. *Using Z: specification refinement and proof*. Prentice Hall, 1996.

[134] A. Zakinthinos and E.S. Lee. The composability of non-interference. In *Proceedings of the 8th IEEE Computer Security Foundations Workshop*, pages 2–8. IEEE Computer Society Press, 1995.

[135] A. Zakinthinos and E.S. Lee. How and why feedback composition fails. In *Proceedings of the 9th IEEE Computer Security Foundations Workshop*, pages 95–101. IEEE Computer Society Press, 1996.

[136] A. Zakinthinos and E.S. Lee. A general theory of security properties and secure composition. In *Proceedings of the 18th IEEE Symposium on Research in Security and Privacy*, Oakland, CA., 1997. IEEE Computer Society Press.